Honeywell
Honeywell Information Systems Italia

Via Laboratori Olivetti - Pregnana Milanese

UNIVERSITA'
DEGLI STUDI DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE
Via G. Colombo, 49
20133 Milano - Tel. 2772234

29/30

NOTE DI SOFTWARE

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

Honeywell
Honeywell Information Systems Italia

Note di software.
è un bollettino trimestrale, edito a cura del Centro di Ricerca e Progettazione della Honeywell Information Systems Italia e del Dipartimento di Scienze dell'Informazione dell'Università di Milano.

Note di software.
intende costituire uno strumento per la rapida e informale circolazione di informazioni, idee ed esperienze nell'area della tecnologia del software. In particolare, intende stimolare il dibattito sulle metodologie orientate alla diminuzione del rapporto costo/qualità dei programmi, e sugli strumenti atti ad automatizzare l'attività di produzione del software in ogni suo aspetto.

La collaborazione a **Note di software** è libera.
In particolare, si sollecitano contributi nella forma di brevi monografie e bibliografie essenziali commentate sui seguenti temi: tecniche e strumenti per la programmazione strutturata e modulare, metodologie e strumenti per il testing e il debugging dei programmi, documentazione del software, aspetti organizzativi e gestionali nella costruzione di sistemi software di grandi dimensioni.

Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Comitato di Redazione (HISI - Via ai Laboratori Olivetti - Pregnana Milanese), o di inviare direttamente il dattiloscritto in triplice copia.
I contributi non dovrebbero superare le quattromila parole.

I lavori accettati per la pubblicazione vengono revisionati dal Comitato di Redazione, che si avvale del contributo di specialisti del settore. Ogni contributo pubblicato riflette, comunque, le opinioni dei suoi autori, e non necessariamente quelle del Comitato di Redazione. Eventuali revisioni saranno effettuate di comune accordo fra il Comitato di Redazione e i proponenti.

Note di software.
Viene distribuito ad aziende, Enti e specialisti del settore che ne facciano richiesta alla redazione.

Direttore responsabile: Paolo Ghelfi
Comitato di Redazione: Roberto Polillo - Wanda Anselmetti - Stefania Bandini
Redazione: Franco Soresini

29/30

NOTE DI SOFTWARE

Dicembre 1985

Indice:

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

QUESTO NUMERO...

Questo numero di NOTE DI SOFTWARE ospita una raccolta di articoli, curata da Andrea Granelli, sul progetto DLT 16000, nato da una collaborazione tra l'Istituto di Cibernetica e il Dipartimento di Scienze e Tecnologie Biomediche dell'Università di Milano. Da tale progetto sono nati non solo il prototipo di un personal computer, ma anche numerosi sviluppi software, sia di base che per applicazioni specifiche. La collezione di articoli qui raccolti offre una panoramica che tocca tutti questi aspetti.

La Redazione

Questo numero doppio di NOTE DI SOFTWARE è dedicato a Giulio Dossi, valente ricercatore e promotore del progetto DLT. Il suo apporto, sia nella fase di definizione, che nella fase iniziale della realizzazione prototipale, è stato fondamentale. Egli, infatti, univa alle doti di acuto e meticoloso ricercatore una grande carica umana che ha avuto un importante potere coesivo e motivante sul gruppo da cui è partita la sperimentazione.
La sua scomparsa immatura, non ha comunque impedito che il progetto si concretizzasse. Le sue proposte ed i suoi consigli sono stati seguiti e si sono man mano arricchiti di contributi nuovi.

Scopo del progetto era la realizzazione di un personal computer 32-bit potente e a basso costo gestito da un software molto flessibile.
La raccolta di lavori che pubblichiamo vuole essere una testimonianza sul significato del progetto e sulla sua concreta realizzazione.
Il resto della storia è ben rappresentata nella seguente raccolta di articoli: ricercatori e studenti con entusiasmo si sono dedicati ad attività di ricerca utilizzando questo computer.
Questo numero di NOTE DI SOFTWARE raccoglie una larga parte di queste esperienze attualmente in corso.

Preludio a questa raccolta una breve introduzione di G. Degli Antoni in cui sono messi in risalto gli aspetti più motivazionali del progetto e in particolare, il problema del software d'autore.

Il primo articolo, "Principi architetturali dell'elaboratore DLT", di A. Granelli, riassume la storia del progetto e descrive la struttura hardware e software del DLT, soffermandosi sui principi architetturali.

Successivamente in "Un sistema operativo ed un interprete multitask e multiwindow su personal computer" G. Colli e L. Spampinato descrivono l'implementazione di un interprete LISP dotato di funzionalità avanzate quali la gestione di più finestre e la possibilità di eseguire concorrentemente più processi.

Nel terzo lavoro "Architettura e implementazione del compilatore LISP in ambiente incrementale per personal computer" M. Cassinadri, S. Crispiatico e L. Spampinato presentano il progetto e la realizzazione di un compilatore LISP.

Il quarto articolo di A. Granelli, "Grafica di Base su DLT", presenta la implementazione di un gruppo di librerie grafiche. Alcuni DLT sono infatti dotati di una scheda grafica che viene completamente controllata da un gruppo di procedure.

La parte di collegamento del DLT con la rete locale 3M installata nei laboratori didattici del Corso di Laurea in Scienze nell'Informazione, è stata invece curata da C. Garavaglia e G.P. Rossi, che nel loro articolo "A communication system providing transport services on a local network of microcomputers" descrivono in particolare l'implementazione del livello di trasporto OSI.

L'articolo di A. Granelli, D. Marini, S. Missaglia "Implementazione dello standard grafico del CORE" descrive invece la realizzazione di un linguaggio grafico evoluto, il CORE proposto dall'ACM Siggraph, che permette il trasporto di programmi grafici su differenti computers.

L'implementazione di alcuni algoritmi per l'analisi delle Reti di Petri è stata invece curata da D. Pieragostini nell'articolo "Algoritmi di analisi per Reti di Petri".

G. Pasi e M. Cividati, nel loro articolo "Un sistema integrato di consultazione clinica e gestione dati per un centro ustioni", descrivono la creazione di uno strumento per la gestione dei dati clinici in un centro ustioni.

Infine D. Boschetti, nel suo articolo "Gestione di un archivio di immagini come ausilio alla refertazione radiologica", descrive la realizzazione di uno strumento di ausilio alla refertazione radiologica, che utilizza il DLT in configurazione grafica ed il Mouse.

A. Granelli

LO SVILUPPO SOFTWARE IN AMBITO UNVERSITARIO

Gianni Degli Antoni
Istituto di Cibernetica - Università di Milano

Chi si accinge ad analizzare il rapporto fra Università e software si trova di fronte una notevole massa di problemi di solito accuratamente ignorati dalla letteratura e dalla pubblicistica in genere.
Si deve intanto constatare con chiarezza che la natura del software sta cambiando con la diffusione di macchine a basso costo capaci di impiegarlo.

Una tale diffusione trasforma qualsiasi software, specie se di buona qualità, in potenziale prodotto culturale o professionale essendo ogni giorno più difficile una distinzione fra queste due ultime entità.
La comparsa delle case editrici di libri di testo del mercato del software in varie forme (dischetto associato a testi, libri elettronici di varia foggia, dischetti e nastri come prodotti) ha trasformato definitivamente il software in una entità da leggere ed impiegare attraverso un irrilevante computer.
Inutile citare alcuni prodotti di successo, in questo mercato: è più importante realizzare che il software impone una revisione radicale di cosa sia scuola, università, professione, organizzazione del lavoro e proprietà del risultato del proprio lavoro, nel caso questo sia attuato con computer.

A proposito di questo ultimo aspetto occore confrontare chi lavora con computer e chi lavora senza computer per ottenere certi risultati.
Chi lavora senza fornisce come risultato del proprio lavoro ciò che gli è stato chiesto e per cui viene retribuito. Naturalmente la sua personale esperienza si è arricchita. Di tale esperienza nessuno potrà avvalersi direttamente se non chi la ha acquistata: costui finirà con il disporne secondo le sue aspirazioni.

La situazione è ben differente nel caso di chi impiega il computer. Infatti costui avrà ottenuto due lavori: uno è il risulato richiestogli. L'altro sono una serie di procedure che materializzano una parte della sua esperienza su un qualche supporto fisico. Va da se che si pone il problema della proprietà di questo secondo risultato anche se sembra difficile negare a chi ne affermasse la proprietà la validità delle sue tesi, anche se la discussione verrà spostata sulla proprietà del supporto (dischetto, carta, eccetera).
È un fatto che è impossibile imporre a chiunque di consegnare il secondo risultato, a meno che sia proprio quello il dovuto: ma naturalmente a quel punto il problema si ripropone per un successivo livello!

In ambito universitario il problema è da tempo presente: ricercatori che stampano tabelle usando programmi da loro progettati, risultati ottenuti e programmi mai consegnati nè a qualche pubblicazione nè consegnati a quale si voglia ufficio.
Non che sempre la pubblicazione di quei programmi abbia una grande importanza in ogni caso: no, certo! Tuttavia una politica su questo tema può permettere molte cose che vanno dalla crescita del settore software con riduzione di importazione nel settore di software per cui esiste la precisa competenza, alla creazione di incentivi.

Questo punto è estremamente importante poichè se l'Università non riesce a trattenere le persone che hanno capacità a produrre questa nuova cultura in TUTTE LE DISCIPLINE si troverà ben presto di fronte ad una gigantesca crisi che è ben evidenziata dalla fuga di ottimi elementi verso strutture produttive nel settore delle discipline informatiche, che è solo il primo per ragioni ovvie ad essere sottoposto a questa nuova situazione. La riduzione del prezzo dei computer farà lo stesso in TUTTE LE ALTRE DISCIPLINE, anche in quelle apparentemente più lontane dal mondo produttivo come conseguenza della trasformazione della editoria certacea in editoria elettronica.

Ecco una delle ragioni per una Università di creare una cultura di software interna e dei meccanismi di trasferimento del risultato del lavoro proprio e dei propri componenti (fra cui importantissimi gli studenti), senza che ciò avvenga senza sotterfugi certo difficili da scoprire ed anche da identificare, sotterfugi che comunque hanno realmente impedito la presenza in Italia di cose elementarissime nel settore del

software, in nome forse della importanza scientifica dei risultati. Ma in verità per la mancanza di coraggio nel rendere realmente pubbliche attività che sono state svolte e per le quali è troppo complesso il meccanismo di costruire leciti incentivi, a cui non è assolutamente giusto rinunciare.

Esempi sono molti e non valgono solo per le Università: infatti ad esempio tutti i prodotti di buona qualità nel settore della progettazione automatica presto o tardi potranno diventare software di facile impiego da immettere nel mondo della didattica: perchè questo avviene poco o male? Non è così in USA. Laggiù chiunque si trova una miriade di strade per valorizzare con qualche fatica SIGNIFICATI di altri lavori svolti. Perchè non dovrebbe avvenire anche qui? Il non farlo ha un ovvio significato: sperpero di risorse che spesso vengono dedicate alla realizzazione di pacchetti software che verranno impiegati pochissime volte!

Certo occorre una cultura della qualità, una cultura della interfaccia uomo macchina, una cultura delle applicazioni, una cultura delle conseguenze sociali del software: culture difficili da costruire e che debbono muoversi in mille insidie sia per le difficoltà dei temi che per la mancata comprensione di quell'immenso fenomeno che è il software nelle sue svariate realizzazioni!

Presso l'Istituto di Cibernetica, che durante la sua breve vita ha prodotto e ceduto (per contratto) del software, il problema è stato più volte analizzato anche se ancora non è chiara una soluzione che soddisfi tutti i requisiti. La nascita di un personal proprio tuttavia facilita una line di tendenza. Su quel software una frazione degli studenti realizzerà o sta realizzando la propria tesi. Il risultato del lavoro è proprio loro ma anche dell'Istituto che ha messo a disposizione le risorse necessarie. E qualora il contributo di docenti e ricercatori sia importante il risultato del lavoro sarà anche in misura appropriata di docenti e ricercatori.

È chiaro che lo studente una volta laureato potrà usare le competenze acquisite liberamente e senza vincoli. Ma non potrà utilizzare direttamente e liberamente il risultato del proprio commercializzandolo direttamente. Sarà tuttavia suo interesse diffonderlo e coinvolgere in questo le strutture universitarie così come sarà interesse delle strutture universitarie trovare delle strade per commercializzare direttamente i risultati di buona qualità (che non dimentichiamolo sono solo conoscenza meccanicamente viva) ottenuti nel prioprio ambito.

La presenza di un proprio personal ha molti vantaggi: i risultati possono essere pubblicamente distribuiti senza che siano direttamente utilizzabili su altre macchine; i trasporti implicano investimenti e sono di solito difficili da effettuare senza la competenza di chi ha realizzato il pacchetto sofware. Insomma è possibile ottenere pubbliche azioni e incentivi corretti!

Bene o male è quanto avviene in Università USA anche se talvolta in USA sono controversie legali cause fra enti che sfruttano i risultati ed università che li hanno prodotti. Le Università in tempi recenti sono alquanto più attente a questi aspetti. Ma se ciò non fosse lecito o possibile la cultura informatica sarebbe rimasta alquanto arretrata: occore ora un salto di qualità: la consapevolezza dei fenomeni segnalati ed il coraggio di costruire strutture libere nell'ambito di strutture vincolate. È forse il solo modo con cui si potrà impedire una nuova fuga di cervelli.

PRINCIPI ARCHITETTURALI DELL'ELABORATORE DLT

Andrea Granelli
Istituto di Cibernetica - Università di Milano
Dipartimento di Scienze e Tecnologie Biomediche - Università di Milano

RIASSUNTO

Questo articolo descrive le caratteristiche hardware e software del computer DLT. Viene anche illustrata per sommi capi la storia del progetto che ha condotto alla realizzazione di questo computer.

ABSTRACT

The present paper describes the hardware/software principles used to build the DLT computer. But before that, the history of the projext is briefly told.

1. INTRODUZIONE

In questo articolo mi propongo di raccontare brevemente la storia del progetto, mettendo in risalto il contributo dei partecipanti al progetto e soffermandomi sui risultati ottenuti.
In particolare esaminerò i principi architetturali del DLT, la sua configurazione hardware e il suo sistema operativo.
Si farà riferimento, in questo articolo, alla configurazione base del DLT senza considerare i modelli con la scheda grafica.
La realizzazione del sistema grafico viene infatti descritta in un altro articolo "GRAFICA DI BASE PER IL DLT", sempre presente in questa raccolta.

2. LA STORIA DEL PROGETTO

L'idea di costruire un computer nacque nei primi mesi del 1982 dopo una serie di incontri tra due realtà universitarie, il Dipartimento di Scienze e Tecnologie biomediche dell'Università degli Studi di Milano presso l'ospedale S. Raffaele e l'Istituto di Cibernetica, sempre della stessa Università.
La motivazione principale era quella di creare e fare crescere un gruppo di lavoro interdisciplinare che acquisisse precise competenze.

Il gruppo iniziale era composto dalle seguenti persone: Gianni Degli Antoni e Massimo Luzzana erano i due coordinatori del progetto e rappresentavano le due strutture universitarie. David Berger era il responsabile del software ed in particolare del sistema operativo ed dell'interprete. Giancarlo Caramenti insieme a Fabio Confalonieri curava la parte di progettazione della scheda hardware ed era coadiuvato da Gian Paolo Rossi e da Giuseppe Rossi, che in particolare hanno curato l'ingegnerizzazione del circuito stampato. Giulio Dossi, a cui è dedicata questa raccolta di articoli, era il principale promotore del progetto; il suo compito era curare l'interfaccia tra hardware e software e il coordinamento operativo tra i vari membri dell'equipe. Infine il sottoscritto era il responsabile del sofware grafico sviluppato per gestire la scheda grafica e di una parte del sistema operativo (il monitor sulla EPROM).

Durante l'evoluzione del progetto, il gruppo si è arricchito di molti contributi; tra questi si possono ricordare Fabio Cottini e Giorgio Marino che hanno curato l'estensione del compilatore a 32 bit (quello originario era a 16-bit) e l'interfacciamento con strumentazione biomedicale di vario genere, mentre altri vengono testimoniati dalla presente raccolta di articoli.

Il nome DLT venne dopo (per un certo periodo il computer venne da alcuni chiamato IPPOCRATE visto che una delle sue destinazioni era l'ambiente

medico) e fu motivato dal fatto che l'ingegnierizzazione del computer venne fatta dalla Delta Instrumentation srl.

Torniamo comunque alla storia. Il primo passo fu quindi decidere quale microprocessore utilizzare e la scelta cadde sul NS 16032 della National Semiconductor. Questo componente non era ancora disponibile sul mercato europeo e quindi fu fatta venire una piastra di sviluppo direttamente dai centri di ricerca della National Semiconductor a Santa Clara in California. Erano le sue caratteristiche di elevata integrazione e la sua upward compatibility che ebbero un ruolo decisivo nella scelta di questo microprocessore; infatti oltre a fornire una architettura molto compatta e sofisticata, dotata tra l'altro di una gestione efficiente di più stacks e di funzioni ad alto livello come DIV e MOD (divisione e resto su variabili di tipo intero), garantiva la totale compatibilità software con il futuro NS 32032, il microprocessore full-32 bit della National Semiconductor.
Si poneva allora un quesito fondamentale: dove ingegnerizzare il computer? Si scelse una piccola ditta, la Instrumentation srl, che così entrò nel progetto.

Il Pascal come linguaggio di sviluppo fu scelto invece per la presenza al Dipartimento di Scienze e Tecnologie Biomediche, di David Berger; era infatti tra quelli che, sotto la guida di Ken Bowles, avevano implementato l'UCSD Pascal; inoltre i sorgenti di questa versione, sviluppata presso il campus di San Diego della California University (UCSD), e cioè la 2.0, erano liberamente disponibili nei circuiti universitari.
Oltre a ciò, il sistema operativo UCSD Pascal è stato ideato per garantire la trasportabilità del software; infatti è costruito sopraa un simulatore (detto P-Machine) che interpreta un linguaggio simbolico detto P-code. Il problema della trasportabilità del sistema, si riconduce alla trasportabilità della P-machine, che ha dimensioni molto ridotte.
La scelta del Pascal fu poi rafforzata dalle caratteristiche architetturali del microprocessore della National Semiconductor; esso fornisce infatti una serie di funzioni che semplificano sensibilmente l'implementazione della P-Machine.

Si è così realizzato un computer totalmente sotto controllo (si poteva cioè intervenire per modificare qualsiasi sua caratteristica, sia hardware che software), che si è quindi caratterizzato come prezioso strumento didattico, permettendo tra l'altro di adattarlo (sia con modifiche hardware che software) alle esigenze che via via sorgevano.
E proprio in questa ottica è stato possibile modificare il Monitor (il programma residente sulle EPROM che fornisce le routines di base per l'I/O e carica il sistema operativo), rendendo quindi possibile lo sviluppo di un software di comunicazione che ha permesso di collegare il DLT alla rete locale 3M, in funzione presso i laboratori Didattici del corso di Laurea in Scienze della Informazione (si veda l'articolo di C. Garavaglia e G.P. Rossi sempre in questo numero).

3. LA ARCHITETTURA

3.1 Caratteristiche hardware

La configurazione hardware del DLT 16000 è composta da tre moduli principali. Un alimentatore, due floppy disk drivers da 5" 1/4 doppia faccia, doppia densità (circa 820 kbytes formattati ciascuno) ed il modulo di CPU. Quest'ultimo è basato sul microprocessore a 16 bit 16032 della National Semiconductor, su 16 kbytes di EPROM (Erasable Programmable Read Only Memory) e su tre circuiti di interfaccia: una interfaccia seriale (RS-232C) tripla per collegare il video, le stampanti ed eventuali periferiche remote; una interfaccia basata sul WD 2793 per il Floppy Disk Controller ed infine una interfaccia parallela per l'espansione del sistema (la scheda grafica ad alta risoluzione oppure il disco Winchester).
Nella figura 1 è illustrato lo schema dei componenti presenti sulla piastra.
L'utilizzo delle EPROM, che contengono il Monitor del sistema al posto della tradizionale ROM è stato motivato dalla loro riprogrammabilità. Ciò ha permesso di aggiornare o adattare il Monitor riutilizzando gli stessi componenti.

La memoria centrale è di 128 kbytes, di cui 64 kbytes vengono utilizzate dall'interprete e i suoi dati, e le restanti 64 kbytes dal programma Pascal correntemente in uso.

Per quanto riguarda le periferiche collegabili con il DLT, si è deciso di non porre alcuna restrizione. Per questo motivo sono presenti nel sistema tre interfacce seriali, le cui caratteristiche (baude rates, parity, on-off...) sono modificabili mediante un modulo del sistema operativo, il System Configurator. Sono quindi collegabili tutte le periferiche dotate di un interfaccia seriale RS232C. In particolare anche il video viene collegato tramite un'interfaccia seriale.

3.2 Architettura del microprocessore

La architettura del microprocessore NS 160032 include 16 registri presenti nella CPU e illustrati nella figura 2. di questi ci sono 8 general-purpose per facilitare le esigenze di memorizzazione temporanea ad alta velocità; questi registri sono di 32 bit e possono essere usati anche in coppia. I rimanenti sono invece dedicati; vediamone rapidamente l'utilizzo da parte del microprocessore:

PC: è il Program Counter e contiene sempre l'utilizzo del primo byte dell'istruzione correntemente eseguita; viene incrementato solamente quando l'istruzione corrente è stata com-

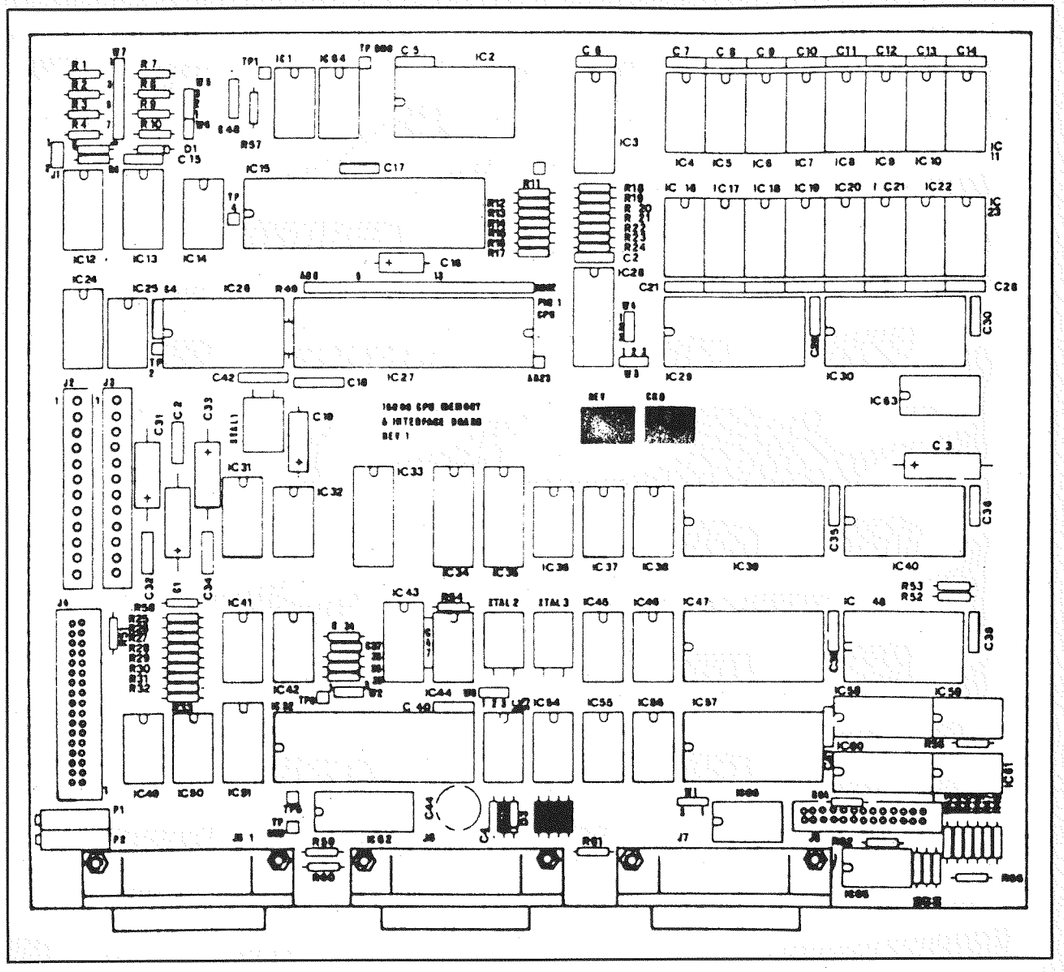


Fig. 1 Schema Hardware del DLT

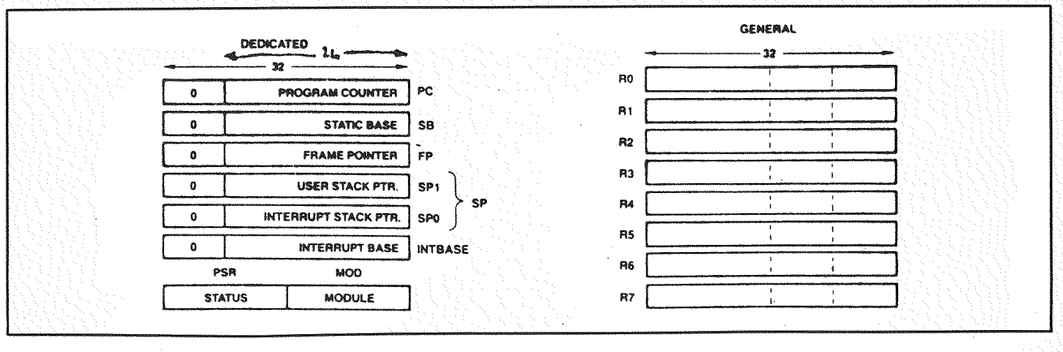


Fig. 2 Registri del NS 16032

pletata

SP: registro usato per la gestione dello stack; a seconda dello stato in cui è il microprocessore (bit S in PSR) questo registro gestisce l'INTERRUPT STACK (e viene chiamato SP0) oppure l'USER STACK (e viene chiamato SP1)

FP: il registro Frame Pointer viene usato dalle procedure per accedere ai parametri e alle variabili locali sulla stack; FP punta cioè all'area di dati creata dinamicamente all'inizio della procedura e detta "Activation Record"

SB: lo Static viene invece usato come registro base per accedere alle variabili globali in un programma ed è una copia del campo SB dell'elemento corrente della Module Table

INTBASE: questo registro contiene l'indirizzo della dispatch table per le interruzioni e le traps; questa tabella contiene i descrittori delle procedure di gestione delle interruzioni/traps.

MOD: il registro Module permette l'utilizzo di routines esterne al modulo correntemente eseguito; punta sempre infatti all'elemento corrente della Module Table entry; questa tabella è composta da elementi di 16 bytes composti da:
SB: (Static Base) punta all'area statica (per le variabili globali) del modulo
LB: (Link Base) punta alla Link table del modulo
PB: (Program Base) punta alla prima istruzione del codice del modulo

PSR: Il Processor Status Word contiene i codici di stato per il microprocessore; e di 16 bit ed è diviso in due parti da 8 bit; gli 8 bit meno significativi sono accessibili a tutti i programmi, mentre gli altri 8 sono utilizzabili solo da programmi eseguiti in supervisor mode

Per estendere la potenzialità del microprocessore esiste, nella Control Section della CPU, un registro di 4 bit detto CFG (Configuration Register) che dichiara la presenza di alcuni chip esterni. Viene utilizzato solo da una istruzione (SETCFG) che deve essere eseguita solamente all'accensione del computer.
I chip collegabili alla CPU sono:

FPU: Floating Point Unit per accelerare le operazioni matematiche in virgola mobile

MMU: Memory Management Unit per gestire processi concorrenti

Custom Slave Processor: chip disegnato appositamente per soddisfare particolari problemi

La memoria centrale è uno spazio di indirizzamento lineare ed uniforme; le locazioni di memoria sono numerate sequenzialmente da 0 a (2^24-1) e contengono un byte (8 bit). Con l'unica eccezione della Page Tables usate per il Memory management, non c'è l'obbligo di allineamento per operandi con una dimensione maggiore del bytes (word, double-word, quad-word). Ciò vuol dire che operandi di qualsiasi lunghezza possono iniziare ad ogni "byte address". È chiaro però che, per aumentare il throughput, conviene generalmente allineare i dati.

In figura 3 viene rappresentato il formato generale di una istruzione per il NS 16032. La "Basic Instruction" è composta da uno e tre bytes e contiene l'Opcode (mnemonico che identifica il tipo di operazione da eseguire) e fino a a due modi di indirizzamento generale da cinque bit ciascuno. Dopo il campo Basic Instruction, vi è la parte opzionale dell'istruzione la cui presenza dipende sia dall'Opcode che dal tipo di indirizzamento selezionato. Questo campo può contenere registri indice, displacements (offset da aggiungere ai valori presenti nei registri) oppure metri (immediate value).

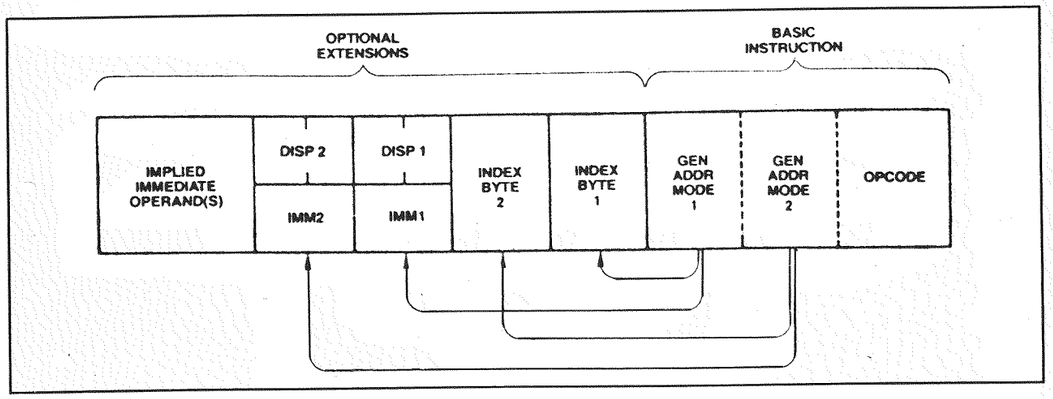


Fig. 3 Formato generale di una istruzione

Concludiamo questa breve panoramica sulla architettura del NS 16032 con le modalità di indirizzamento utilizzabili che permettono di avere un quadro più completo sulle sue potenzialità operative.

REGISTER: l'operando è disponibile in uno degli 8 registri general-purpose.

REGISTER RELATIVE: un registro general-purpose contiene un indirizzo al quale viene aggiunto un displacement per ottenere l'indirizzo effettivo dell'operando in memoria.

MEMORY SPACE: è identico al precedente solo che i registri sono quelli dedicati.

MEMORY RELATIVE: i registri dedicati contengono un indirizzo di memoria che contiene a sua volta un indirizzo che aggiunto al displacement genera l'indirizzo effettivo.

IMEMDIATE: l'operando è codificato all'interno dell'istruzione.

ABSOLUTE: l'indirizzo dell'operando è specificato da un campo displacement nell'istruzione.

EXTERNAL: si legge un indirizzo da un elemento specifico della Link Table corrente; a questo indirizzo viene aggiunto un displacement, ottenendo l'indirizzo effettivo.

TOP OF STACK: lo stack pointer correntemente selezionato (SP0 o SP1) specifica la locazione dell'operando sul quale viene eseguita una operazione di PUSH o POP a seconda che si debba scrivere o leggere.

SCALED INDEX: benchè sia codificato come una modalità di indirizzamento, Scaled Indexing è una opzione per quasi tutti i tipi di indirizzamento; aggiunge all'indirizzo effettivo calcolato in uno dei modi precedentemente descritti il valore di un registro general-purpose moltiplicato per 1, 2, 4 o 8; serve per implementare la struttura CASE del linguaggio Pascal.

3.3 Il software di base

Il software di sistema standard è composto di nove moduli e segue la struttura del sistema operativo UCSD, di cui ne è una derivazione (anche se ampiamente modificata):

SISTEMA OPERATIVO: sistema operativo monoutente basato sull'UCSD Pascal System.

SCREEN EDITOR: un editor full-screen disegnato per la creazione sia di programmi Pascal che di documenti generici.

FILER: programma che permette di gestire un File System lineare.

PASCAL COMPILER: compilatore a un passo basato sul compilatore P2 sviluppato all'ETH di Zurigo da Wirth.

LINKER: programma che permette l'unione, in un unico programma, di più moduli compilati separatamente.

LIBRARIAN: programma per la gestione di più librerie composte da moduli compilati separatamente.

DISK FORMATTER: permette di formattare i floppy disk ed il winchester secondo diverse modalità (single/double density...)

SYSTEM CONFIGURATOR: permette di modificare diversi parametri di sistema, in particolare quelli legati alle interfacce con le periferiche (baude-rates, parità...)

Questi moduli, tutti scritti in Pascal, non sono eseguiti direttamente dal microprocessore, ma vengono interpretati da un altro modulo detto INTERPRETE (e memorizzato nel sistema col nome SYSTEM.DELTA). Ciò è alla base del concetto di P-machine; essa infatti, è un computer virtuale, frapposto tra il sistema operativo ed il microprocessore. Per ciò i programmi Pascal non sono scritti per il NS 16032, ma per la P-machine; quindi il compilatore non genera Assembler NS 16032 bensì un Assembler virtuale, detto P-code. Questo meccanismo, pur comportando una certa inefficienza in fase di esecuzione dei programmi, ha il grande vantaggio che rende il sistema software facilmente portabile. Sarà infatti sufficiente modificare opportunamente solo l'interprete (e ovviamente la parte a livello più basso, che è anche essa scritta in Assembler e contiene il Bootstrap ed il BIOS (Basic I/O Subsystem)) senza dover modificare il compilatore. Infatti questa particolare architettura software separa nettamente la fase di controllo sintattico (parsing fatto dal compilatore) dalla fase di generazione del codice (fatta dall'interprete).

Inoltre l'architettura del microprocessore si è rivelata molto versatile nella realizzazione/modifica dell'interprete per la P-machine. Sono infatti disponibili delle istruzioni P-code come per esempio DIV e MOD (divisione e resto interi) oppure meccanismi molto flessibili per la gestione della stack. E questo è molto importante in quanto va ricordato infatti che la P-machine e una "stack-machine" cioè una macchina a 0 indirizzi.

4. PROSPETTIVE FUTURE

Allo stato attuale il DLT è disponibile in molte configurazioni. La memoria va da un minimo di 128 kbytes ad un massimo di 521 kbytes. Per la memorizzazione sono disponibili sia i floppy disks doppia faccia/doppia densità (1 megabyte formattato) che il disco rigido. Per la versione con la scheda grafica sono disponibili anche delle periferiche in input, in particolare mouse e tavoletta digitalizzatrice.
Il sistema operativo implementa una P-machine a 32 bit, che permette di indirizzare tutti i 512 kbytes, mentre con la versione precedente (utilizzata dalla maggior parte degli autori di questa raccolta di articoli) si potevano indirizzare esclusivamente 64 kbytes.

È stato anche sviluppato una versione differente di monitor che permette la creazione di due processi concorrenti che si scambiano informazioni mediante un mailbox e permette il controllo da computer di una apparecchiatura elettromedicale.

È inoltre in fase di sperimentazione un software applicativo distribuito funzionante su rete locale per la gestione di reparti sanitari.

5. BIBLIOGRAFIA

[1] Jensen, Wirth: PASCAL USER MANUAL AND REPORT, II edition, Springer Verlag, 1975.

[2] Wirth, ALGORITHMS + DATA STRUCTURE = PROGRAMS, Prentice Hall, 1976.

[3] NS 16000: INSTRUCTION SET REFERENCE MANUAL, National Semiconductor Corporation, pub. no. 420010099-001.

[4] NS 16000: PROGRAMMER'S REFERENCE MANUAL, National Semiconductor Corporation, pub. no. 420306565-001PB.

[5] Clark, Koehler, THE UCSD PASCAL HANDBOOK, Prentice Hall, 1982.

[6] Pascal System Delta D.1 PROGRAMMER'S MANUAL.

[7] Pascal System Delta D.1 USER'S MANUAL.

[8] Zelitzky, Srivastava, COMPILERS ON THE NS 16000, Toronto Conference Proceedings of "USENIX", 1983.

UN SISTEMA OPERATIVO ED UN INTERPRETE LISP MULTITASK E MULTIWINDOW PER PERSONAL COMPUTER

Giuseppe Colli, Luca Spampinato
Istituto di Cibernetica - Università di Milano

RIASSUNTO

Utilizzare il linguaggio LISP su piccoli sistemi con architettura tradizionale comporta spesso problemi di efficienza e di interfaccia utente. Nell'articolo che segue vengono descritte le soluzioni adottate per l'implementazione di un sistema operativo e di un interprete LISP che tengano conto proprio di tali necessità. Il progetto è stato sviluppato sul personal computer DLT 16000, dotato di un processore a 32 bit, come tesi del corso di laurea in Scienze dell'Informazione.

ABSTRACT

LISP programmers on small traditional systems often find problems given by little efficiency and non-user-friendly interfaces. The following notes describe the solutions implemented on an operating system and an interpreter for LISP in which efficiency and user-friendliness come first. The project has been developed on the personal 32-bit computer DLT 16000 in Computing Science course.

1. INTRODUZIONE

Una stazione di lavoro per lo sviluppo di software dedicato all'Intelligenza Artificiale deve garantire, più che una elevata efficienza in termini di tempi di esecuzione, la possibilità di tenere sotto controllo contemporaneamente il comportamento delle singole parti del programma in fase di realizzazione e la struttura delle interfacce, sia di quelle tra operatore e programma, che di quelle che consentono la comunicazione tra i vari sottoprogrammi.

I sistemi che attualmente garantiscono tali prestazioni sono di solito progettati appositamente e quindi di alto costo a causa della complessità dello hardware e del software di basso livello.

Sulla base di queste considerazioni è nata l'idea di trasformare un personal computer con architettura tradizionale in una stazione di lavoro totalmente dedicata al LISP, linguaggio "storicamente" utilizzato per programmi di Intelligenza Artificiale.
Il progetto, concepito all'Istituto di Cibernetica dell'Università di Milano, è stato sviluppato come tesi di laurea del corso di Scienze dell'Informazione, sul Personal Computer DLT-16000 della DELTA Instrumentation, che si basa su una CPU NS16032 della National Semiconductors.

I primi risultati di questo progetto sono contenuti in [4] (per ciò che riguarda il sistema operativo) e [9] (che descrive la struttura dell'interprete), entrambe svolte in collaborazione con il dott. L. Spampinato dell'Istituto di Cibernetica.

Nelle due tesi citate sono descritti il disegno e l'implementazione di una stazione di lavoro LISP multitask-multiwindow; in particolare nella prima vengono definite le funzioni di gestione dei processi, delle finestre e delle condizioni eccezionali, mentre l'altra è dedicata all'interprete LISP e alla gestione della memoria.

L'architettura dell'interprete tiene conto dell'esigenza di fornire primitive di facile utilizzo per la creazione e la gestione di più processi in un ambiente multiwindow; inoltre, sono stati curati particolarmente quegli aspetti già in partenza penalizzati dalla scelta del micro elaboratore, cioè il tempo impiegato nella valutazione delle espressioni e l'occupazione della memoria, che, inizialmente limitata a 128K, è stata successivamente estesa a 512K.
Si sono introdotti quindi particolari meccanismi di gestione di tutto il sistema, vale a dire:

a) eliminazione della "case analysis" sul tipo delle

espressioni da valutare, con l'adozione dei puntatori tipati (che cioè contengono già al proprio interno le informazioni necessarie all'interprete per il loro riconoscimento);

b) paginazione della memoria, che consente un trattamento omogeneo delle strutture dati associate a tutti i tipi di oggetti LISP;

c) garbage collector incrementale (ispirato a quello di H. Baker [3]);

d) trattamento di finestre e processi come strutture dati (oggetti) LISP, quindi manipolabili direttamente dall'interprete, che in tal modo non risulta separato dal sistema operativo;

e) interpretazione delle funzioni tail-ricorsive in modo completamente iterativo;

f) estensione di sintassi e semantica delle tradizionali primitive di trattamento delle condizioni eccezionali (catch e throw).

Infine tutto il lavoro è stato condotto in vista degli sviluppi successivi, comprendenti il compilatore, il file system e l'introduzione di un gestore di memoria virtuale.

2. I PUNTATORI TIPATI

Gli atomi LISP, costituenti gli elementi base del linguaggio, sono stati suddivisi in tre categorie, a seconda del loro "modo" di essere valutati:

- atomi normali: sono atomi definiti dall'utente;
- atomi primitivi: sono i "nomi" delle funzioni primitive LISP (car, cdr, cons, ecc);
- atomi speciali: sono i "nomi" delle espressioni speciali LISP, la cui valutazione cioè non è standard (per esempio cond, quote, lambda, ecc.).

Gli atomi normali sono descritti da un record di tre campi:

- un puntatore alla stringa di caratteri che rappresenta l'atomo (P_NAME);
- un puntatore al valore corrente;
- un puntatore alla property list.

Il metodo di valutazione degli atomi normali risulta quindi nella semplice lettura del campo valore corrente.

Gli atomi primitivi hanno anch'essi una rappresentazione a tre campi:

- un puntatore alla stringa;
- un puntatore a se stesso (poichè il valore di una funzione primitiva è la funzione stessa);
- un puntatore alla routine (in linguaggio macchina) che implementa l'applicazione della funzione primitiva agli argomenti, che si suppongono già valutati. Questa routine rappresenata il "metodo" di applicazione dell'atomo.

Tre campi anche per gli atomi speciali:

- un puntatore alla stringa;
- un puntatore al metodo di valutazione;
- un puntatore al metodo di applicazione.

Un esempio di quest'ultimo caso è fornito dall'atomo "lambda".
Infatti una lista che cominci con (lambda...) ha per valore se stessa, mentre una lista del tipo ((lambda...) arg1 arg2...) ha come valore il risultato della applicazione della funzione (lambda...) agli argomenti che la seguono.

Questa scelta permette di estendere facilmente l'insieme delle funzioni residenti nell'interprete, oltre ad aumentare l'efficienza nella valutazione delle espressioni LISP. Infatti, la valutazione di una qualsiasi espressione consiste nell'esecuzione della routine assembler associata al primo atomo della lista.

Il rispetto di alcune convenzioni sull'uso dei registri inerni della macchina (che non vengano qui illustrate in dettaglio) consente infine di aggiungere metodi di valutazione in maniera indipendente dagli altri.

Altri tipi di puntatori sono quelli che si riferiscono a numeri, stringhe, finestre e processi. Questi due ultimi tipi, di cui si parlerà in seguito più in dettaglio, vengono così trattati in maniera sintatticamente omogenea agli altri.

3. LA MEMORIA A PAGINE E IL GARBAGE COLLECTOR

Oltre all'efficienza in termini di tempo di esecuzione si è reso necessario curare l'ottimizzazione dell'occupazione di memoria, in considerazione della ridotta capacità della macchina su cui è stato realizzato il progetto e in generale delle macchine per uso personale.

A tale scopo si è adottata una gestione della memoria a due livelli. Il primo prevede la suddivisione della memoria di lavoro in pagine di 512 bytes, legate l'una all'altra in una tipica free-list, mentre al secondo livello opera il garbage collector.

Tale gestione permette un uso più flessibile ed un recupero più efficace dello spazio di memoria: la paginazione permette infatti di trattare in modo omogeneo tutte le strutture dati associate ai vari oggetti presenti nel sistema (finestre, processi, ecc.), mentre il garbage collector, operante sulla memoria a pagine,

provvede al recupero delle aree occupate dalle strutture dati LISP semplici (liste, atomi, numeri e stringhe).

Per ridurre ulteriormente l'utilizzo di memoria si è scelto inoltre di rappresentare le liste LISP in CDR-notation (per i cui dettagli vedi [2] e [3]) e di recuperare in maniera automatica (in fase di collezione) le aree di memoria occupate dagli atomi che non siano più utilizzati all'interno delle strutture dati esistenti, e quindi non siano più "vivi" nel sistema.

Il collettore utilizzato è una variante "a pagine" del garbage collector incrementale descritto in [3]; esso prevede l'utilizzo di due aree di memoria separate, denominate FROM e TO.
La zona TO è quella da cui vengono prelevate nuove celle di lavoro, mentre la zona FROM viene periodicamente visitata e da essa vengono copiate nella zona TO le celle ancora utili, cioè con almeno un riferimento "vivo". Nel momento in cui la zona FROM è stata completamente visitata (cioè sono state copiate tutte le celle utili), essa viene rilasciata e la zona TO diventa a sua volta zona FROM, da collettare, ricominciando il processo sopra descritto. Quest'ultima è la fase detta di "flipping", in cui le due zone cambiano di funzione e di utilizzo.

4. FINESTRE E PROCESSI

Un aspetto di flessibilità del sistema è dato dal trattamento dei processi e delle finestre come strutture dati direttamente "visibili" dall'interprete; in tal modo è possibile far svolgere ad esso compiti ordinariamente affidati al sistema operativo, quali la creazione, l'accesso, la modifica e, al termine della loro "vita", la rimozione di tali oggetti.

Un processo viene creato per mezzo di una espressione del tipo:

(process ‹expr› ‹priority›)

dove ‹expr› è l'espressione che viene valutata durante la vita del processo stesso e ‹priority› è un intero che indica per quanti cicli-macchina il processo va lasciato attivo prima di una sospensione. L'espressione può essere un loop infinito, nel qual caso il processo può venir cancellato solo per mezzo di una

(killp ‹process›)

eseguita da un altro processo (sono infatti vietati i suicidi); oppure la valutazione dell'espressione può avere una terminazione, e allora il suo valore finale può venire assegnato utilizzato se il processo in precedenza era stato assegnato come valore ad un atomo.

Dal punto di vista implementativo, un processo è un pagina di memoria al cui puntatore viene assegnato il tipo "processo" e in cui sono contenute tutte le informazioni necessarie a creare un ambiente autonomo in cui viene valutata l'espressione associata; in altre parole viene allocata una zona di memoria (o meglio più zone non necessariamente contigue, trattandosi di pagine) in cui possano essere contenute le informazioni relative alla valutazione in corso e il cui accesso sia consentito solo al processo da cui sono utilizzate; cioè vengono allocate (sottraendole alla free-list) e legate alla pagina contenente il descrittore del processo le pagine di memoria necessarie a contenere gli stack necessari all'interprete.

Parte essenziale del sistema operativo è il gestore dello scheduling dei processi, il cui algoritmo è una variazione adattata all'ambiente LISP del classico time-sharing, esteso in modo da evitare conflitti sulla memoria comune e situazioni di deadlock. Vengono fornite all'utente primitive di sospensione, sincronizzazione e risveglio esplicito dei processi, che permettono la creazione e la gestione di coroutines, grazie alla condivisione della memoria tra tutti i processi. La primitiva di interruzione è

(wait)

che interrompe il processo finchè non venga risvegliato esplicitamente da parte di un altro processo attivo; il risveglio può essere effettuato tramite una

(wakeup ‹proc›)

che rimette in coda ready il processo ‹proc›, mantenendo attivo il processo che la effettua, oppure tramite una

(goto ‹proc›)

che invece sospende il processo che la esegue e passa il controllo al processo ‹proc›.

In maniera analoga una finestra è vista dal sistema come un puntatore ed un oggetto LISP, il cui tipo è "finestra", ed è rappresentata da una pagina di memoria, su cui opera un editor, parametrizzato rispetto alle dimesioni della finestra; il buffer di editing viene creato allocando le pagine di memoria necessarie e legandole alla pagina della finestra, che funziona, come nel caso del processo, come punto di partenza di tutto lo sviluppo dell'oggetto.

Il risultato delle scelte architetturali descritte è la possibilità di disporre di un sistema estremamente flessibile ed adattabile alle esigenze dell'utente direttamente a run-time.

Infatti, poichè gli oggetti di tutti i tipi vengono allocati nella stessa zona di memoria, è facoltà dell'utente scegliere come occuparla; egli cioè può scegliere tra più alternative:

- editare su molte finestre, utilizzando grandi buffer, e attivare invece un numero minimo di processi concorrenti;

- creare un grande numero di processi che operano su poche finestre e svolgono "poco" lavoro ciascuno (cioè causano l'allocazione di poche pagine per gli stack di valutazione);

- avere poche finestre e pochi processi, che però svolgono una grande quantità di valutazione, destinando quindi la maggior parte dello spazio di memoria agli stack;

All'utente viene fornito un insieme di funzioni LISP a basso livello che permettono la creazione, la sincronizzazione e la rimozione di processi indipendenti nonchè la creazione, la gestione e la distribuzione di finestre, su ognuna delle quali è attivo l'editor di cui già si è detto.

L'editor è completamente integrato nel sistema e consente la contemporaneità della stesura e della esecuzione dei programmi, sia sulla stessa finestra sia su altre finestre, risultando quindi di particolare utilità in un ambiente di sviluppo LISP.

La struttura dati che rappresentano le finestre e i processi sono trattate dal sistema come oggetti LISP, consentendone quindi l'accesso da parte dell'utente per mezzo di funzioni primitive del linguaggio, e infine il recupero (da parte del garbage collector) qualora non siano più raggiungibili.

Il sistema operativo permette inoltre il trattamento dell'input/output da console tramite funzioni LISP di basso livello, parametriche rispetto alla finestra su cui agiscono ed è inoltre già predisposto per la gestione di memorie di massa.

5. LE FUNZIONI TAIL-RICORSIVE

Un'importante caratteristica dell'interprete è quella di ridurre le funzioni tail-ricorsive a comportamenti iterativi. In un interprete privo di questa possibilità, la valutazione di una espressione come

```
(member 'C '(A B C))
```

dove member è definita come:

```
(lambda (el lis)
(cond ((null lis) nil)
((eq (car lis) el) t)
(t (member el (cdr lis))) ))
```

richiederebbe l'allocazione di 3 frame nello stack delle variabili del sistema (environment), in cui verrebbero contenute le coppie:

```
el – C
lis – (A B C)
---------------
el – C
lis – (B C)
---------------
el – C
lis – (C)
```

I primi due frames però non hanno più alcuna utilità nel momento in cui risulta vera (eq (car lis) el), perchè il valore della funzione è indipendente dai valori precedentemente assunti dalle variabili ed e lis. Inoltre non occorre richiamare ricorsivamente la valutazione di

```
(member el (cdr lis))
```

ma è sufficiente continuare la valutazione allo stesso livello e nello stesso ambiente di prima, fatti salvi le opportune variazioni dei legami delle variabili ed e lis.

L'innovazione dell'interprete sta appunto nel riconoscere i legami e le chiamate ricorsive inutili e quindi nell'evitare di effettuare push in sovrannumero sullo stack.
Il controllo risulta iterativo e i nuovi legami vengono sovrascritti su quelli vecchi e non più aggiunti al top dello stack. Tale operazione, pur allungando (di poco) i tempi di binding, consente di risparmiare spazio e di utilizzare senza problemi un loop infinito come espressione legata al processo iniziale:

```
(toplevel default)
```

dove toplevel è

```
(lambda (_w) (print_w (eval (sysread_w)))
(toplevel_w)
```

6. LE CONDIZIONI ECCEZIONALI

Il sistema operativo fornisce le primitive di basso livello per il trattamento delle condizioni eccezionali. Allo scopo sia di raccogliere l'eredità di ZISP, interprete LISP per Z-80, [16] sia di fornire all'utente strumenti per il controllo dei "casi particolari" che siano il più possibile in linea con i recenti sviluppi dei linguaggi di programmazione (vedi ADA), sono state estese la sintassi e la semantica della tradizionale coppia LISP catch e throw:

```
(catch (errname1 handler1)
(errname2 handler2)
.....
‹expr›) (throw ‹errname› ‹mess›)
```

Tramite queste funzioni l'utente può riconoscere, correggere ed eventualmente gestire non solo le eccezioni da lui stesso segnalate, ma anche i possibili errori sintattici e semantici che vengono segnalati dall'interprete durante la normale valutazione delle espressioni LISP. Infatti catch segnala al sistema che l'errore errname-i eventualmente trovato durante la valutazione di ‹expr› andrà gestito per mezzo dell'handler-i associato; a sua volta throw segnala il riconoscimento di un errore definito all'utente ‹errname› e passa il controllo all'handler associato all'errore, trasmettendogli anche il messaggio ‹mess› valutato nell'ambiente in cui l'errore viene riconosciuto. Un throw implicita viene inoltre eseguita ad ogni errore di sistema rendendo quindi l'errore gestibile da una catch dell'utente, in cui venga associata una funzione all'errore systerr.

7. CONCLUSIONI

Le primitive di gestione di finestre e processi qui descritte, pur non fornendo un vero e proprio sistema operativo completo, permettono all'utente di costruire per mezzo di sole funzioni LISP un proprio sistema operativo ad alto livello. Il compilatore e le primitive di gestione della memoria di massa, già in via di realizzazione, costituiscono infine le necessarie aggiunte per ottenere una completa work-station dedicata allo sviluppo di programmi in LISP.

8. BIBLIOGRAFIA

La bibliografia che segue comprende, oltre ai riferimenti agli articoli e alle tesi citati nel testo, la letteratura utilizzata durante la stesura di queste ultime.

[1] A.V. Aho, J.D. Ullman, PRINCIPLES OF COMPILER DESIGN, Addison Wesley, 1977

[2] J. Allen, ANATOMY OF LISP, McGraw-Hill, 1978

[3] H.G. Baker, LIST PROCESSING IN REAL TIME ON A SERIAL COMPUTER, CACM 21-4 (Apr. 1978) p. 280-294

[4] G. Colli, UN SISTEMA OPERATIVO LISP MULTITASK E MULTIWINDOW PER PERSONAL COMPUTER, Tesi di laurea in Sc. dell'Inf. Univ. di Milano, A.A. 1983-84

[5] P. Greussay, ITERATIVE INTERPRETATION OF TAIL-RECURSIVE LISP PROCEDURES, Dip. di Inform., Univ. di Vincennes, Sett. 1976

[6] A.N. Habermann, INTRODUCTION TO O.S. DESIGN, SRA, 1976

[7] P. Henderson, FUNCTIONAL PROGRAMMING: APPLICATIONS AND IMPLEMENTATION, Prentice Hall, 1980

[8] J.E. Hopcroft, J.D. Ullman, FORMAL LANGUAGES AND THEIR RELATIONS TO AUTOMATA, Addison Wesley, 1969

[9] S. Messelodi, UN INTERPRETE LISP IN AMBIENTE MULTITASK-MULTIWINDOW PER PERSONAL COMPUTER, Testi di Laurea in Sc. dell'Inf., Univ. Stat. Milano, A.A. 1983-84.

[10] L. Siklossy, LET'S TALK LISP, Prentice Hall, 1976

[11] G.L. Steele Jr., LAMBDA: THE ULTIMATE DECLARATIVE, MIT AI Lab, AI Memo 379, Nov. 1976

[12] G.L. Steele Jr., G.J. Sussman, LAMBDA: THE ULTIMATE IMPERATIVE, MIT AI Lab, AI Memo 353, Mar. 1976

[13] G.L. Steele Jr., G.J. Sussman, LAMBDA: THE ULTIMATE OPCODE, MIT AI Lab, AI Memo 514, Mar. 1979.

[14] G.L. Steele Jr., G.J. Sussman, SCHEME: A DIALECT OF LISP, MIT AI Lab, AI Memo 452, Gen. 1978

[15] G.J. Sussman, LISP, PROGRAMMING AND IMPLEMENTATION, IN FUNCTIONAL PROGRAMMING. AN ADVANCED COURSE, Cambridge Press

[16] ZISP, Manual Utenti, 1982

ARCHITETTURA E IMPLEMENTAZIONE DEL COMPILATORE LISP IN AMBIENTE INCREMENTALE PER PERSONAL COMPUTER

M. Cassinadri, S. Crispiatico - Dipartimento di Scienze dell'Informazione - Università di Milano
L. Spampinato - Quinary s.r.l.

RIASSUNTO

Nel corso degli ultimi anni è nata, in ambito universitario, l'idea di trasformare un personal computer in una stazione di lavoro per lo sviluppo di applicazioni di Intelligenza Artificiale, basata sul linguaggio LISP. I primi risultati del progetto sono stati la realizzazione di un sistema operativo e di un interprete, già descritti in un precedente articolo di G. Colli e L. Spampinato dal titolo "UN SISTEMA OPERATIVO E UN INTERPRETE LISP MULTITASK E MULTIWINDOW PER PERSONAL COMPUTER". Il passo successivo è stata la realizzazione di un compilatore completamente integrato con il resto del sistema. L'articolo che segue descrive l'architettura del compilatore e il suo inserimento nel sistema. Il progetto è stato sviluppato come tesi di laurea del corso di Scienze dell'Informazione nell'A.A. 1984/85.

ABSTRACT

In the course of the last years the idea of turning a p.c. into a work station for the development of A.I. applications based on LISP has risen in the university sphere. The first results of the project have been the carrying out of an operating system and an interpreter, already described in a previous article (see [1]). The next step has been the carrying out of a compiler fully integrated with the rest of the system. This article describes its architecture and its introduction into the system. The project has been developed in Computing Science in 1985.

1. INTRODUZIONE

La stesura del compilatore si inserisce in un progetto di più ampia portata che prevede anche lo sviluppo di un sistema operativo e di un interprete quali componenti il kernel base di una stazione di lavoro per il LISP; tale stazione di lavoro potrà in seguito divenire un valido strumento per la trattazione di problemi di Intelligenza Artificiale.
L'idea di procedere alla progettazione di un compilatore, infatti, nasce dalla prospettiva di poter realizzare applicazioni di una certa complessità nel campo dell'I.A.; quando un particolare programma è stato definitivamente testato e si presenta nella sua forma eseguibile, diventa utile poter disporre di un compilatore che permetta di utilizzare il software prodotto nella sua versione più "efficiente".

Il compilatore è stato realizzato in modo da risultare completamente integrato con il resto del sistema, intendendo con ciò una serie di aspetti che verranno illustrati nel seguito dell'articolo.
Il progetto, concepito all'Istituto di Cibernetica dell'Università Statale di Milano, è stato sviluppato come tesi di laurea del corso in Scienze dell'Informazione, su un Personal Computer DLT 16000 della DELTA Instrumentation, che si basa su una CPU NS1632 della National Semiconductors.

I risultati ottenuti nella fase di sviluppo del sistema operativo e dell'interprete sono contenuti rispettivamente nelle tesi di [2] e di [3], mentre le descrizioni relative ad architettura e implementazione del compilatore si trovano rispettivamente in [4] e [5].
In particolare nelle prime due vengono definite le funzioni di gestione dei processi, delle finestre e delle condizioni eccezionali, nonchè la struttura dell'interprete LISP e la gestione della memoria. Per una descrizione di questi argomenti si veda il già citato articolo [1]. Nelle ultime due, oggetto di questo articolo, vengono descritte le caratteristiche del compilatore (architettura, tecniche utilizzate per il trattamento della symbol table e della tail-ricorsione nel codice intermedio, il linguaggio intermedio, l'integrazione del compilatore nel sistema) e la sua implementazione (modifiche alla versione precedente dell'interprete per tenere conto del compilatore, trattamento dei compilati come oggetti del sistema, fase di loading del codice intermedio).

L'architettura del compilatore è caratterizzata dal modo in cui vengono compilate le espressioni del linguaggio LISP; si noti che la generazione del codice intermedio avviene attraverso funzioni scritte interamente in LISP.
Agli atomi primitivi e speciali è associato un "metodo di compilazione" cioè una funzione scritta in LISP, quindi compilabile anch'essa, che rappresenta il modo in cui deve essere (ed effettivamente viene) compilata un'espressione LISP avente un atomo di questo tipo come primo elemento.

Al momento di compilare un'espressione LISP che abbia un atomo speciale o primitivo come funzione da applicare (p. es. (car '(a b c)), l'applicazione della procedura che ne costituisce il metodo di compilazione produce una sequenza di istruzioni in un codice intermedio; tale sequenza, una volta tradotta nel linguaggio della macchina e caricata in memoria, all'atto dell'esecuzione realizzerà le azioni previste dalla semantica dell'espressione compilata.

Il codice che si ottiene dalla compilazione di un'espressione costituisce un oggetto di tipo "compilato" (che va ad aggiungersi agli altri tipi di oggetti trattati dal sistema) e può essere assegnato come valore a un atomo definito dall'utente, in modo da poter poi essere utilizzato. L'informazione che stabilisce l'appartenenza di un oggetto alla categoria "compilati" risiede, come per gli altri oggetti del sistema, nel campo informazioni contenute nel puntatore all'oggetto stesso.
Il codice compilato consiste in una sequenza di istruzioni nell'assembler del DLT 16000 che realizzano le azioni dettate dalla funzione definita dall'utente, quando vengono eseguite run-time.

Il dispatcher del compilatore (cioè la procedura che decide le azioni da intraprendere per la compilazione dell'espressione in esame) non è incentrato, come avviene nei compilatori per il LISP più tradizionale, su una "case analysis" sull'espressione da compilare che decida quale azione intraprendere in base all'entità del primo elemento di tale espressione.

Infatti, grazie all'introduzione del concetto di puntatore "tipato" e all'utilizzo di una tabella come descritto in [1], è possibile procedere alla compilazione delle varie sottoespressioni componenti l'espressione globale reperendo l'informazione sul loro tipo direttamente nel campo informazioni del puntatore al descrittore dell'oggetto. Il dispatcher del compilatore utilizza lo stesso schema dell'interprete per manipolare le varie componenti dell'espressione da trattare.

2. INTEGRAZIONE INTERPRETE-COMPILATORE

Le scelte effettuate per l'architettura del compilatore si integrano in maniera naturale con la filosofia dell'interprete.
Infatti, così come la valutazione di espressioni tramite interprete avviene con l'utilizzo dei "metodi di valutazione" e i "metodi di applicazione", così la compilazione di espressioni si basa sui "metodi di compilazione" e sui "metodi di traduzione"; così come la valutazione di un'espressione non prosegue mediante "case analysis", allo stesso modo si comporta il compilatore sull'espressione da compilare, sfruttando i puntatori "tipati".
Infine, così come l'interprete, anche il compilatore è stato progettato per trattare funzioni tail-ricorsive in modo da non sprecare spazio in memoria e da accorciare i tempi di esecuzione.

L'oggetto di tipo compilato può essere riferito allo stesso modo in cui si riferiscono gli altri oggetti del sistema.
Un compilato, alla pari di un'espressione interpretata, può essere assegnato come valore a un atomo normale; in questo modo, un atomo definito dall'utente può assumere come valore indifferentemente un oggetto interpretato e uno compilato.

Le funzioni definite dall'utente possono venire compilate indipendentemente l'una dall'altra e l'ambiente non fa alcuna differenza tra funzioni compilate e funzioni interpretate. Le funzioni definite e poi fatte compilare vengono ad aggiungersi alle altre funzioni del sistema in modo naturale e sono utilizzabili allo stesso modo, vale a dire senza dover adottare un diverso comportamento per il fatto che sono dei compilati.
Da questi argomenti si comprende come gli oggetti compilati costituiscano semplicemente uno dei possibili tipi di oggetti che l'interprete deve trattare e, in quanto tali, non richiedono una trattazione diversificata dagli altri tipi di oggetti.

Dal punto di vista dell'utente, non c'è alcuna differenza tra applicare una funzione compilata o la corrispondente funzione interpretata, salvo che naturalmente il compilato è molto più efficiente in termini di tempo di esecuzione; la miglior velocità di esecuzione è più evidente al crescere della complessità della funzione in esame e al crescere della dimensione all'input alla funzione stessa.

Nei termini precisati, i compilati si integrano comple-

tamente nella struttura generale del sistema. È importante sottolineare il fatto che il compilatore sfrutta, fin quando gli è possibile, le potenzialità dell'interprete; ciò significa che il codice compilato fa riferimento alle routines dell'interprete quando deve compiere delle azioni che sono già messe a disposizione tra le primitive dell'interprete. In aggiunta a questo, un'espressione soggetta a interpretazione può fare riferimento all'interno della sua definizione a oggetti di tipo "compilato" e, viceversa, si può eseguire il compilato di un'espressione che riferisca oggetti definiti via interprete; infine, così come una funzione definitiva via interprete può riferire altre funzioni interpretate, anche una funzione compilata può richiamare al suo interno altri compilati.

3. CRITERI IMPLEMENTATIVI

La struttura dati del compilatore è rappresentata dal programma da compilare, che nel nostro caso può essere una qualsiasi espressione LISP. Il compilatore ha il compito di scorrere il programma sorgente costruendo passo per passo il programma intermedio che, una volta tradotto nel linguaggio della macchina e caricato in memoria, potrà essere eseguito realizzando i compiti specificati dal sorgente. Pertanto la produzione del codice eseguibile avviene in due fasi:

il compilatore svolge il suo compito in un'unica passata, generando un codice intermedio scritto in un linguaggio appositamente definito;

il compilatore genera un codice nella sua forma finale di intermedio, provvedendo alla sistemazione dei riferimenti esterni effettuati dall'interno dell'espressione LISP da compilare e al trattamento di eventuali tail-ricorsive nell'espressione stessa; per "riferimento esterno" si intende la presenza nell'espressione di qualsiasi oggetto che si possa considerare come una variabile utilizzata dal sorgente.

4. IL NUCLEO DEL COMPILATORE

Il compilatore realizzato è in grado di compilare qualsiasi espressione LISP sintatticamente corretta. Si comprende bene come il poter compilare qualsiasi espressione fornisca un livello di flessibilità notevole, evitando all'utente di dover mettere la funzione che vuole compilare in una forma accettabile per il compilatore e permettendogli quindi di concentrarsi sul contenuto della funzione stessa, piuttosto che sulla forma.
Il nucleo del compilatore è costituito da una serie di procedure LISP che hanno il compito di intraprendere diverse azioni a seconda della sintassi dell'espressione da compilare, proprio come fa l'interprete quando si trova a dover interpretare una certa espressione.

Il tipico comando che l'utente darà per compilare una funzione avrà il formato seguente (l'atomo "comp" è di tipo speciale);

(set 'a (comp ‹exp›))

Si tratta cioè di assegnare a un atomo normale (a) il compilato della funzione prescelta (‹exp›), in modo da poter in seguito utilizzare l'atomo "a" avente come valore un oggetto di tipo compilato.
Ovviamente, si può anche eseguire direttamente un compilato su degli argomenti senza assegnarlo ad alcun nome di atomo normale:

((comp ‹exp›) arg^1.. arg^n)

L'analogia di comportamento con l'interprete è strettissima; anche un'espressione interpretata, infatti, può essere assegnata come valore a un atomo normale oppure eseguita direttamente sui suoi argomenti senza darle un nome. Al livello più alto la compilazione distingue due casi a seconda del tipo di espressione da compilare:

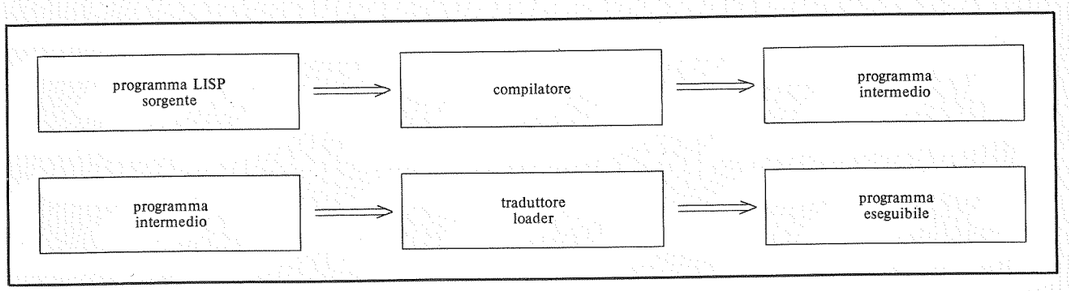


Fig. 1

1) è un oggetto "atomico": se non è un atomo normale (cioè un atomo primitivo, un numero, una stringa,...) la compilazione è equivalente ai fini del risultato finale della sua valutazione per mezzo della funzione eval dell'interprete. Altrimenti, occorre procedere alla sua compilazione vera e propria, in quanto tale atomo potrebbe assumere un qualsiasi valore che è noto solo run-time.

2) è una lista: le azioni da intraprendere sono diverse a seconda che la lista sia una lambda-espressione o meno. Infatti, l'applicazione run-time del compilato di una lambda necessita di una fase preliminare di binding dei parametri formali della lambda ai valori attuali; quindi la fase di compilazione deve produrre una sequenza di istruzioni che, eseguite run-time, realizzino il corpo della lambda in un ambiente in cui i parametri formali e attuali sono stati legati. Se l'espressione compilata non è una lambda, non si devono effettuare legami prima di eseguire il compilato e quindi verrà eseguita l'applicazione del primo elemento della lista (car = funzione da applicare) ai restanti elementi (cdr = argomenti della funzione).

Nessun altro tipo di espressione può essere oggetto di compilazione, poichè in LISP gli oggetti sono atomi o liste.

4.1 Metodi di compilazione

Ad ogni atomo speciale e primitivo è associato un metodo di compilazione, costituito da una funzione LISP che, applicata all'espressione avente come car un atomo di questo tipo, produce una sequenza di istruzioni intermedie atte a ricalcare le azioni svolte e dall'interprete nella valutazione di tale espressione.

4.2 La symbol table del compilatore

Nell'espressione da compilare possono comparire i nomi dei diversi oggetti trattati dal sistema, come atomi normali, numeri, ecc... . Quando il compilatore incontra tali numeri registra la loro presenza in una apposita tabella (che in LISP è una lista), nota come "symbol table" del compilatore (o "tabella dei riferimenti esterni" o "tabella dei rimbalzi").

4.3 La tail-ricorsione nell'intermedio

Una funzione tail-ricorsiva è tale che la sua ultima azione consiste nella chiamata ricorsiva a se stessa oppure a un'altra funzione, con un diverso valore dell'argomento (o degli argomenti). In altri termini, il valore della funzione di partenza sul suo argomento è biunivocamente determinato dal valore della funzione chiamata sull'argomento modificato.

È quindi causa di spreco di memoria e di tempo di esecuzione il salvare l'ambiente relativo alla funzione chiamante; l'interprete gestisce questa situazione evitando di mettere come punto di rientro dalla procedura chiamata la routine che ripristina l'ambiente, se a questo è già stato provveduto una volta, e andando a scambiare i valori delle variabili direttamente in symbol table.

Nel codice intermedio, se la chiamata a una funzione è l'ultima azione da svolgere, si provvede a non rientrare nel codice una volta eseguita tale funzione e a intraprendere sull'ambiente un'azione analoga a quella dell'interprete descritta sopra.

4.4 Il codice intermedio generato

Il linguaggio in cui è scritto il codice intermedio è stato formulato in modo da rispecchiare quelli che sono i meccanismi di funzionamento di un interprete virtuale che disponga degli stessi registri e degli stessi stacks utilizzati dall'interprete reale.
Pertanto, sono state previste istruzioni per la manipolazione di uno stack, istruzioni per operazioni su registri (caricamento, comparazione,...), istruzioni di salti relativi nel codice, istruzioni di salto a routines dell'interprete (con ritorno nel codice compilato o meno).

4.5 Errori di compilazione

Nella generazione del codice intermedio, il compilatore esegue un'analisi sintattica per verificare la correttezza dell'espressione in fase di compilazione.
Nell'intento di seguire il più possibile la filosofia del sistema, si è scelto di generare degli errori compile-time in una forza simile a quelli generati dall'interprete in caso di valutazione di un'espressione sintatticamente scorretta. L'interprete emette un messaggio di errore del tipo:

systerr n

dove "systerr" indica che è stato commesso un errore di sistema e n è un numero che rappresenta il tipo di errore commesso. Il compilatore genera un messaggio di errore analogo, del tipo:

comperr n

dove "comperr" segnala che è stato commesso un errore di compilazione e n è il codice che indica il tipo di errore.

4.6 Traduzione e caricamento del codice intermedio

Quando viene compilata un'espressione vengono prodotte due liste: una contiene quelli che sono definiti "riferimenti esterni" e una che contiene il codice intermedio. Il compito di tradurre e caricare queste due liste in memoria è svolto dalla funzione "transload", scritta in assembler, la cui sintassi è:

sintassi: (transload ‹code-list› ‹ref-list›)

e il risultato è quello di ritornare l'oggetto di tipo compilato, che può essere utilizzato in modo del tutto uniforme agli altri oggetti presenti nel sistema. È possibile costruire quindi liste di compilati, legare questi oggetti ad atomi normali, ecc.
La traduzione del codice intermedio e il caricamento in memoria viene fatto con la stessa filosofia adottata nell'interprete. Ad ogni istruzione del linguaggio ad alto livello è stato associato un "metodo di traduzione".

Ogni metodo è costituito da direttive di traduzione e caricamento che caricano in memoria delle sequenze di bytes, in funzione degli operandi dell'istruzione a cui il metodo è associato. Questi bytes rappresentano la traduzione in linguaggio macchina dell'istruzione in codice intermedio e, una volta eseguiti, realizzeranno le azioni previste dalla semantica dell'istruzione intermedia.
Il codice tradotto infine viene caricato in una zona di memoria riservata a contenere i compilati, in celle contigue.

Il motivo della suddivisione dell'oggetto compilato in codice e riferimenti esterni in questo modo è giustificata dal fatto di poter far diventare radice di collezione la parte di memoria che contiene le variabili.

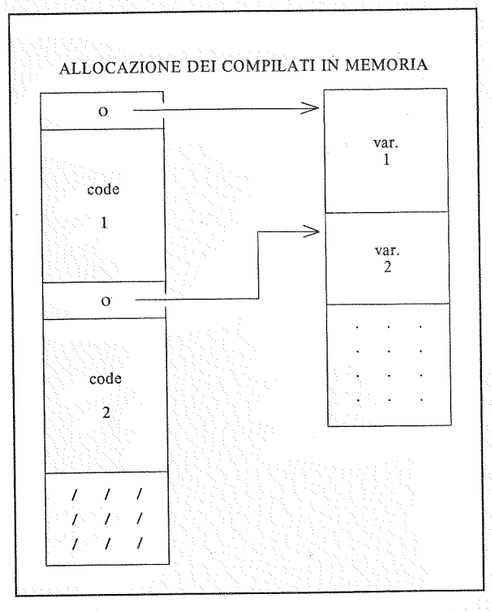


Fig. 2

5. ACCESSO ALLA MEMORIA DI MASSA

L'assenza di un file system penalizzava abbastanza l'uso del sistema; qualsiasi lavoro che infatti veniva svolto, non poteva essere salvato su memoria di massa. In attesa di poter implementare delle primitive per la gestione dei dischi è stata scritta una funzione che carica da disco datafiles, opportunamente trattati, scrivendone il contenuto sulla finestra corrente. Il procedimento per caricare un file è il seguente:

1) Si scrive il testo in un text-file mediante l'editor del sistema UCSD-Pascal che funziona normalmente sulla macchina. Nel testo possono essere inseriti eventualmente commenti: un commento è tutto ciò che segue su una linea dopo il carattere ";". È anche possibile selezionare una o più parti di testo in modo da caricare in memoria solo quella porzione di funzioni che interessano; ciò è possibile marcando il testo con il carattere " ⓐ" in apertura e il carattere "#" in chiusura.

2) Si filtra il text-file prodotto attraverso l'uso di un apposito programma mandando il risultato, che è sottoforma di code-file, sul disco che contiene il sistema LISP-W.

3) Si entra in ambiente LISP e si carica il file con la funzione "load", la cui sintassi è

   sintassi: (load "‹filename›")

   Il risultato di questa operazione, come già accennato, è quello di scrivere il contenuto del codefile sulla finestra corrente consentendo così la valutazione delle espressioni come se fossero state scritte con l'editor LISP.

Aspetto negativo di questo approccio è l'impossibilità di poter aggiornare le e ventuali modifiche apportate alle funzioni caricate rimanendo sempre in ambiente LISP. Qualsiasi modifica deve essere infatti fatta in ambiente UCSD-Pascal ripetendo ogni volta il procedimento sopra descritto.

La quantità di espressioni LISP che possono essere caricate per volta è inoltre limitata dalla dimensione della finestra corrente su cui viene caricato il testo.

Grazie alla modularità di tutto il progetto le modifiche necessarie all'inserimento del compilatore hanno investito una piccola parte dell'architettura del sistema.

Per porter riconoscere e trattare adeguatamente i compilati infatti sono stati introdotti corrispondenti metodi di eval e di apply per tali oggetti. L'introduzione dei metodi di compilazione ha portato a cambiare la struttura degli atomi speciali e primitivi. Sono state inoltre introdotte tutte quelle routine necessarie all'interfaccia tra compilatore e interprete.

6. CONCLUSIONI

L'obiettivo che ci si è posti nella realizzazione del compilatore è stato quello di ottenere un prodotto il più possibile integrato con il resto del sistema, aprendo così la strada alla trattazione di problemi di una certa complessità e allo sviluppo "incrementale" del sistema. Per avere a disposizione una work station più completa da dedicare alla produzione di software in LISP, gli sviluppi necessari sono costituiti dalla realizzazione del file system (potendo così gestire la memoria di massa) ed eventualmente dalla creazione di primitive LISP di basso livello per l'interazione con la memoria, utili per poter scrivere in LISP anche la fase di traduzione e caricamento del codice intermedio.

7. BIBLIOGRAFIA

[1] G. Colli, L. Spampinato, UN SISTEMA OPERATIVO E UN INTERPRETE LISP MULTITASK E MULTIWINDOW PER PERSONAL COMPUTER, articolo già presentato su questa rivista in questo stesso numero

[2] G. Colli, UN SISTEMA OPERATIVO LISP MULTITASK E MULTIWINDOW PER PERSONAL COMPUTER, Tesi di Laurea in Scienze dell'Informazione, Università di Milano, A.A. 1983-84

[3] S. Messelodi, UN INTERPRETE LISP IN AMBIENTE MULTITASK-MULTIWINDOW PER PERSONAL COMPUTER, Tesi di Laurea in Scienze dell'Informazione, Università di Milano, A.A. 1983-1984.

[4] M. Cassinadri, UNA ARCHITETTURA INCREMENTALE PER IL COMPILATORE DEL LISP-W, Tesi di Laurea in Scienze dell'Informazione, Università di Milano, A.A. 1984-85

[5] S. Crispiatico, IMPLEMENTAZIONE DEL COMPILATORE PER IL LISP-W IN AMBIENTE INCREMENTALE, Tesi di Laurea in Scienze dell'Informazione, Università di Milano, A.A. 1984-85

UN SISTEMA DI COMUNICAZIONE CHE FORNISCE SERVIZI DI TRASPORTO PER UNA RETE LOCALE DI PERSONAL COMPUTERS

C. Garavaglia, G.P. Rossi

Istituto di Cibernetica - Università di Milano

RIASSUNTO

Questo articolo descrive un sistema di comunicazione organizzato a livelli disegnato per il trasporto di grandi unità di dati applicativi. I servizi forniti dal sistema di comunicazione sono resi disponibili attraverso la "Communication System Interface" (CSI). La CSI permette l'accesso sia ai servizi del livello di trasporto che alle funzioni di naming. Lo scopo del sistema di trasporto è permettere la comunicazione tra spazi di indirizzi differenti attraverso lo scambio di messaggi. Nella nostra implementazione i messaggi sono insiemi di oggetti non strutturati di dimensione arbitraria e sono trasportati in modo trasparente da uno spazio di indirizzi all'altro. Attraverso la CSI un'entità applicativa può accedere ai servizi "connectionless" disponibili, allo scopo di realizzare ulteriori implementazioni di protocolli di più alto livello. Le entità ai due punti terminali del livello di trasporto possono comunicare riferendosi esplicitamente l'una all'altra utilizzando i loro nomi globli. I nomi globali sono memorizzati in una "name directory" che viene utilizzata da funzioni di naming specializzate, per permettere la descrizione logica dell'ambiente distribuito entro un singolo spazio di indirizzi.

ABSTRACT

This paper describes a layered communication system, supporting the transport of possibility large application data unit. The services provided by the communication system are made visible through the Communication System Interface (CSI) which allows the access to the transport layer services and to the naming facilities.
The purpose of the transport system is the communication amongst different address spaces by passing messages. In our implementation, the message are a collection of possibly large unstructed objects, which are transparently transported from one address space to another. At the CSI and application entity can access connectionless oriented services on purpose provided for supporting further implementation of higher layer protocols. Entities at two end point of the transport layer can communicate by explicitly reby explicitly referencing one another using their global name.
Global names are recorded by a name directory which is accessed by the specialized naming functions for allowing the logical description of the distributed environment within a single address space.

1. INTRODUZIONE

Le stazioni di lavoro autonome e monoprogrammate possono essere interconnesse, utilizzando un mezzo fisico ad accesso multiplo, sia per permettere il trasferimento di files che per condividere risorse comuni.

Questo articolo descrive un sistema di comunicazione appositamente studiato per consentire il trasporto di unità di dati applicativi di dimensione arbitraria tra stazioni di lavoro monoprogrammate connesse da una rete locale.

Il sistema di comunicazione è stato realizzato per: (i) fornire a protocolli di livello più alto i servizi di base di un livello di trasporto, (ii) nascondere alle entità di livello superiore le caratteristiche del particolare sottosistema utilizzato, (iii) offrire ai programmatori delle stazioni di lavoro un insieme di primitive di comunicazione la cui semantica sia simile a quella usata all'interno di ambienti monoprogrammati per il riferimento ad oggetti esterni.

I servizi offerti dal sistema di comunicazione sono visibili attraverso l'Interfaccia al Sistema di Comunicazione (CSI - Communication System Interface), che permette l'accesso sia ai servizi del livello di trasporto che alle funzioni di gestione dei nomi. La CSI è utilizzabile all'interno di programmi applicativi (che da qui in poi verranno chiamati entità applicative o brevemente a-e), attivati su personal computers, per la comunicazione attraverso un sottosistema di comunicazione eterogeneo che usa differenti tecnologie LAN e diversi protocolli d'accesso.

La maggior parte degli sforzi, quando si tratta con reti di stazioni monoprogrammate, è indirizzata a fornire ogni stazione di lavoro con funzioni di comunicazione senza però modificare l'ambiente di programmazione interno.

Ogni processo, nell'ambiente di programmazione considerato, opera entro il proprio spazio di indirizzi e quindi ogni stazione di rete fornisce un unico spazio di indirizzi. Lo scopo del sistema di trasporto è permettere la comunicazione tra spazi di indirizzi differenti trasferendo messaggi. Per quanto riguarda la nostra realizzazione, i messaggi sono insiemi di oggetti non strutturati di dimensione arbitraria e sono trasferiti in modo trasparente da uno spazio di indirizzi all'altro.

Per permettere l'interazione, le entità applicative sono identificate da nomi globali che superano i limiti di ogni spazio di indirizzi. I nomi globali sono raccolti in una directory di rete, e vengono forniti degli strumenti in grado di rappresentarli in un singolo spazio di indirizzi e per permettere il loro riferimento da parte di altri processi.

I servizi di trasporto descritti sono stati pensati per permettere la realizzazione sia di protocolli di file transfer tra coppie di entità ed applicative, che di interazioni client-server per accedere ai servizi offerti da server di rete che amministrano particolari risorse logiche e fisiche.

Questo articolo descrive le caratteristiche di disegno e di realizzazione del livello di trasporto (i cui servizi sono utilizzabili attraverso la CSI) e del livello di rete (che interfaccia i livelli più bassi forniti dal sottosistema di comunicazione). In seguito, quando parleremo dei servizi e delle funzioni di ogni livello, useremo la terminologia suggerita dal modello di riferimento OSI/RM, redatto dall'ISO [1]. Questo è stato fatto per semplicità; infatti, sebbene la suddivisione dei servizi e delle funzioni dei livelli rispecchi quella adottata dall'ISO/RM, i protocolli che sono stati realizzati sono stati disegnati ad hoc per adattarsi a vincoli imposti dal particolare ambiente di programmazione. La versione attuale del sistema di comunicazione descritto è stata realizzata per stazioni di lavoro monoutente e monoprogrammate con sistema operativo UCSD/Pascal, chiamate DLT 16000, [2], connesse alla rete locale Videodata/LAN1 [3], una rete broadband con topologia a bus che utilizza un metodo d'accesso token passing.

I mezzi, utilizzati per realizzare il prototipo del sistema di comunicazione, hanno portato ad alcune scelte implementative che saranno evidenziate in seguito.

2. INTERFACCIA AL SISTEMA DI COMUNICAZIONE (CSI)

Gli utenti di stazioni monoprogrammate sono abituati a conoscere il loro ambiente di programmazione in termini di un singolo processo che opera su oggetti interni ed esterni al programma. Tutte le operazioni riguardanti gli oggetti esterni vengono eseguite in modo sincrono con l'esecuzione del programma, e per qualsiasi riferimento a tali oggetti si utilizzano dei descrittori logici, il cui significato è comunque ristretto al singolo spazio di indirizzi (ad es. i descrittori di files sono usati per identificare univocamente i files). Un sistema di comunicazione, invece, è disegnato per operare entro un ambiente multiprogrammato, e per offrire una visione logica del sistema distribuito, nascondendo alle entità dei livelli più alti sia la loro reale locazione che le caratteristiche del sottosistema di comunicazione [4], [5].

La CSI deve fornire sia il collegamento tra questi due differenti ambienti di programmazione che il modo per accedere ai servizi di comunicazione offerti alle entità di trasporto. Per questo motivo, oltre che per semplicità, abbiamo scelto di raggruppare le procedure che realizzano le funzioni di gestione dei nomi con le procedure che permettono l'accesso ai servizi di trasporto.

Attraverso la CSI, un processo può ottenere un descrittore locale di una porta remota (lpd), che identifica dentro il proprio spazio di indirizzi un partner di comunicazione esterno (questo in analogia con i files, dove si ottengono descrittori di files mediante la procedura di apertura). Inoltre un processo può descrivere se stesso ottenendo tanti descrittori di porte in ricezione (rpd) quanti ne desidera.

I descrittori delle porte sono indici interi utilizzati dalle entità di trasporto per realizzare le funzioni di mapping di identificatori [6]. In seguito, lpd e rpd saranno usati come sinonimi rispettivamente dei parametri indicanti la destinazione e la sorgente nelle procedure di comunicazione.

Tenendo presente le considerazioni precedenti, le 7 primitive, disponibili alla CSI, sono operazioni sincrone e permettono la descrizione del mondo esterno, costituito da processi distribuiti, in termini di descrittori logici (cioè descrittori di porte) con significato locale.

In particolare consideriamo:

(1) send (destinazione, sorgente, ubuff, ndati)

Manda un'unità dati applicativi alla porta "destinazione". Gli "ndati" sono presi dal buffer "ubuff". La primitiva ritorna il numero di dati effettivamente consegnati alla "destinazione". La provenienza dell'unità dati è specificata da "sorgente".

(2) receive (destinazione, sorgente, ubuff, ndati)

Riceve "ndati" provenienti da "sorgente". I dati sono depositati nel buffer "ubuff". È possibile ricevere da una sorgente non specificata se si utilizza il nome locale "0". La primitiva ritorna il numero effettivo di dati ricevuti.

(3) initialize

Inizializza le strutture necessarie alla comunicazione.

(4) assertname (nome globale)

Inserisce un nuovo elemento, corrispondente a "nome globale" nella directory di rete, e ritorna il descrittore della porta in ricezione (rpd) associata a quel nome.

(5) locatename (nome globale)

Ritorna un descrittore locale nella porta remota (lpd) associata al "nome globale".

(6) removename (nome globale)

Rimuove l'elemento corrispondente a "nome globale" nella directory di rete, e toglie tutti i riferimenti locali ad esso ottenuti dai vari processi in rete.

(7) stop

Consente la terminazione di un processo, dal punto di vista della comunicazione, in modo corretto.

In pratica, un'a-e può accedere, attraverso l'Interfaccia al Sottosistema di Comunicazione, sia ai servizi del livello di trasporto, che ai servizi di ricerca e aggiornamento della directory di rete. Mentre le funzioni di comunicazione (1) e (2) sono strumenti generali per accedere ai servizi punto a punto del livello di trasporto, le funzioni di naming (3), (4), (5), (6), (7) offrono il modo per far corrispondere un ambiente distribuito ad un singolo spazio di indirizzi.

3. ARCHITETTURA A LIVELLI E SERVIZI DI TRASPORTO

Il livello di trasporto è stato sviluppato sopra il livello di rete, come mostrato in fig. 1. A sua volta, il livello

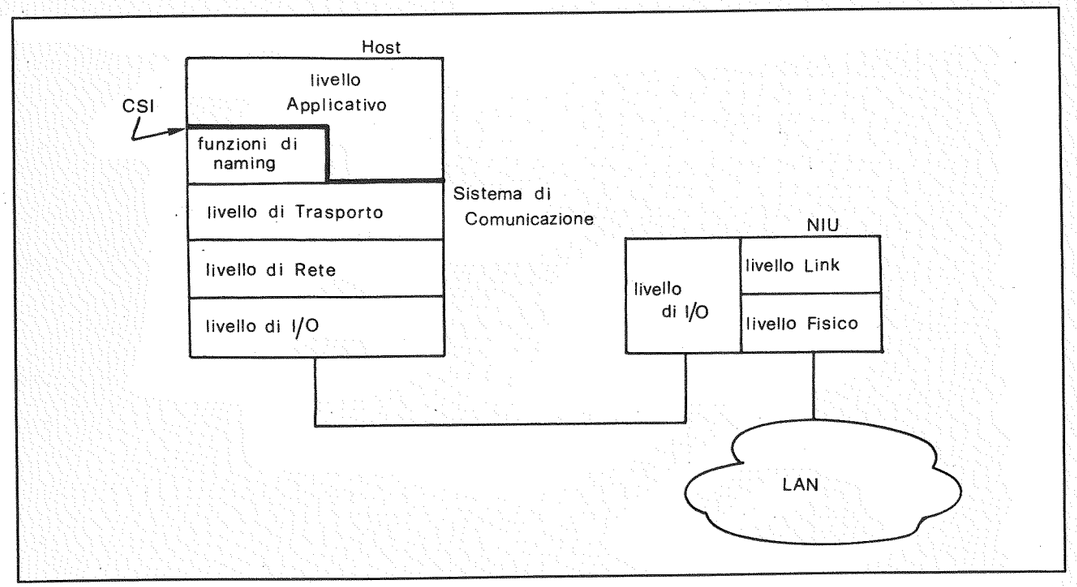


Fig. 1

di rete accede ai servizi offerti dal livello link attraverso quello che abbiamo chiamato livello di I/O. Il livello di I/O fornisce la connessione fisica tra la stazione e l'unità d'interfaccia alla rete (NIU) utilizzando il protocollo RS232. Nella nostra realizzazione, il livello di rete utilizza direttamente le chiamate di input/output fornite dal sistema operativo del DLT 16000.

Il livello di rete effettua le funzioni di frammentazione e composizione delle unità dati e di indirizzamento delle stazioni di rete. Questo livello offre alle entità di trasporto servizi orientati alla connessione; inoltre le connessioni di rete corrispondono in modo biunivoco alle connessioni link [7].

Un indirizzo di rete è specificato dalla coppia ‹n. host› ‹n. porta fisica›. La stessa coppia ‹n. host›, ‹n. porta fisica› identifica un indirizzo a livello link, quindi si potrebbe pensare di evitare che la gestione delle connessioni e delle funzioni di rete sia fatto a livello rete, rendendo così inutile il livello stesso.

Tuttavia, si è deciso di realizzare in ogni stazione un livello di rete principalmente perchè è probabile che l'impiego di differenti sottosistemi di comunicazione richieda, a livello link, differenti formati di indirizzi e diverse procedure d'interazione. Ci è sembrato quindi necessaria l'esistenza di un livello in grado di fornire al livello superiore un'interfaccia trasparente rispetto alla natura del sottosistema di comunicazione. L'attuale interfaccia di rete è stato disegnato per accedere ai servizi orientati alla comunicazione forniti dal livello link della LAN che è stata utilizzata e anche per adattarsi ad alcune sue esigenze.
Considerando le caratteristiche della nostra stazione di lavoro, abbiamo deciso di permettere che al massimo possa essere attiva in ogni istante una sola connessione di rete, infatti le connessioni multiple a questo livello avrebbero richiesto la presenza esplicita di più processi. Quindi l'entità di trasporto richiede, ed eventualmente ottiene, una connessione di rete quando vuole spedire un'unità di dati applicativi e rilascia la connessione al completamento del trasferimento, oppure quando vengono riscontrati errori.

In realtà, l'ambiente monoprogrammato e il fatto che sia permesso mantenere una sola connessione alla volta rendono inutile la gestione di una sola reale connessione di rete, in questo modo si ottiene un'effettiva riduzione di lavoro da parte del sistema. Inoltra, così è possibile propagare i benefici ottenuti dalla connessione link fino al livello di trasporto.

Sebbene non vengano stabilite e rilasciate connessioni di trasporto, e ogni unità dati del livello di trasporto venga trasferita in modo indipendente da un'altra, è possibile offrire agli utenti del livello di trasporto alcune informazioni come il numero di dati realmente trasmessi, e la garanzia della sequenza per le unità trasferite.

Le funzioni principali del livello di trasporto sono:

(1) mapping degli identificatori

Il livello di trasporto deve realizzare una duplice funzione di corrispondenza tra gli identificatori e cioè: (i) locale-remota, per trovare l'rpd del partner in corrispondenza ad un lpd locale; (ii) logico-reale, per fornire al livello di rete il reale indirizzo di rete del partner remoto.

(2) scatter and gather

Questa funzione evita la copiatura in e da aree temporanee del livello di trasporto di unità di dati applicativi, così facendo vengono ridotti oneri per la gestione di protocolli di più alto livello.

(3) notifica delle eccezioni

Quando viene riscontrato un errore, l'entità del livello di trasporto rende visibile nella struttura "csi-exception", a cui si può accedere attraverso la CSI, informazioni che permettono all'entità applicativa di conoscere i motivi del fallimento.

(4) sequenza

Viene rispettato l'ordine delle chiamate alle primitive di trasporto.

(5) controllo sul trasferimento

Questa funzione permette all'entità applicativa di conoscere esattamente il numero di dati effettivamente ricevuti dall'entità remota. È comunque un servizio offerto dal livello di rete.

Si è pensato, inoltre, di permettere alle entità applicative di utilizzare, in modo controllato, dei descrittori "well-known" per indirizzare processi server.
L'idea è di mettere a disposizione, all'interfaccia del sistema di comunicazione, un numero fissato di lpd, che sono automaticamente associati ad indirizzi reali di processi di uso comune. In questo caso, le funzioni di corrispondenza tra gli identificatori sono garantite implicitamente, e vengono stabilite all'atto della configurazione del sistema.

4. IDENTIFICATORI E FUNZIONI DI MAPPING

Per consentire il riferimento e la localizzazione di processi entro l'ambiente distribuito, esistono, a qualsiasi livello della nostra architettura, degli identificatori.
Per meglio dire, si utilizzano i nomi a livello applicativo, i loro descrittori logici a livello di trasporto e gli

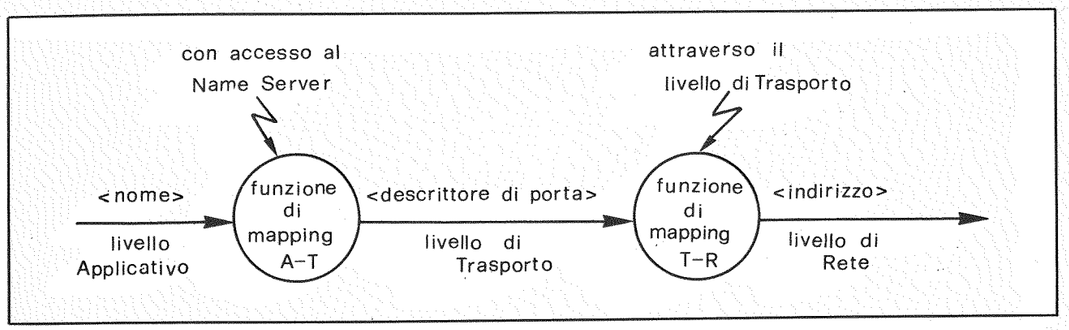


Fig. 2

indirizzi a livello di rete. Utilizzando le funzioni di mapping, le procedure di gestione dei nomi fanno corrispondere gli lpd agli rpd, mentre le entità di trasporto fanno corrispondere i descrittori di porte agli indirizzi reali di rete (fig. 2).

I processi, nell'ambiente distribuito considerato, sono identificati attraverso nomi globali che vengono memorizzati nella directory dei nomi. Una chiamata alla primitiva "assertname" produce l'inserimento di un nuovo elemento nella directory dei nomi e restituisce un rpd. Ogni processo può avere parecchi nomi globali, con gli associati rpd, conversazioni separate (questo sarà utile nel caso in cui si tratti con sottosistemi di comunicazione che consentono connessioni concorrenti a livello link).

Due a-e, prima di poter comunicare, devono esplicitamente "conoscersi", cioè ottenere i rispettivi lpd, in questo modo di stabiliscono le corrispondenze necessarie a livello di trasporto per permettere l'associazione degli indirizzi.

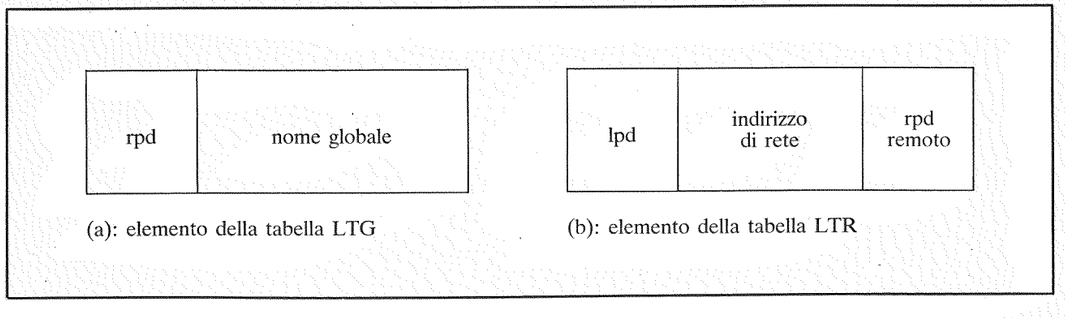
Ogni lpd è il descrittore locale dell'rpd associato a un nome di un partner remoto. Per ottenere un lpd, corrispondente ad un rpd, occorre chiamare la primitiva "locatename" specificando il nome globale del partner di comunicazione.

Una volta ottenuti un lpd e la corrispondenza interna al livello di trasporto di lpd con rpd remoto, questi mantengono significato fino a che l'rpd non è rimosso dal processo che lo ha creato. La rimozione di un rpd può avvenire esplicitamente, tramite la primitiva "removename", oppure esplicitamente, quando cioè vi è una terminazione regolare del processo, vale a dire mediante la primitiva "stop".

La directory dei nomi può essere centralizzata oppure distribuita. La nostra realizzazione considera un directory centralizzata amministrata da una stazione speciale di rete chiamata Name Server. Il Name Server è descritto da un lpd well known all'interfaccia con il livello di trasporto. Un semplice meccanismo di richiesta e risposta con il Name Server viene utilizzato dalle primitive che realizzano le funzioni di gestione dei nomi.

4.1 Le funzioni di mapping del livello di trasporto

Le entità di trasporto forniscono un'associazione tra i descrittori di porte. Per permettere le funzioni di mapping locale-remoto e logico-reale, vengono utilizzate le strutture dati LocalToGlobal (LTG) e LogicalToReal (LTR) (fig. 3).



(a): elemento della tabella LTG

(b): elemento della tabella LTR

Fig. 3

La tabella LTG fornisce la corrispondenza tra il nome globale (o i nomi globali) del processo locale e il suo rpd. È utilizzata per verificare la correttezza del parametro specificato come "sorgente" nelle primitive "send" e "receive". Viene aggiornata dalla primitiva "assertname".

La tabella LTR contiene le informazioni per realizzare le funzioni di corrispondenza tra gli indirizzi. Viene utilizzata da un'entità di trasporto per costruire l'header dell'unità dati del livello di trasporto, come conseguenza di una chiamata alla primitiva "send", e per scopi di controllo quando viene chiamata la primitiva "receive".

Entrambe queste strutture dati vengono aggiornate quando viene richiesta la rimozione di un nome globale.

5. I PROTOCOLLI

Durante la realizzazione, per far fronte ai vincoli posti dall'ambiente di programmazione, sono state fatte alcune scelte che hanno condotto alla definizione di protocolli studiati ad hoc tra entità dello stesso livello. Cioè:

(1) in una stazione monoprogrammata tutte le operazioni di I/O sono sincrone con il programma applicativo. Dal momento che i programmi sono totalmente deterministici, eccetto per quanto riguarda le loro interazioni, occorrerebbe, per garantire una corretta cooperazione, che un processo di ricezione sia attivato mentre il processo di destinazione non ha ancora fatto la sua "receive". Per evitare quindi la perdita di dati provocata dalla mancanza di un processo di ricezione, si è deciso di impedire al processo sorgente di trasmettere informazioni fino a che il processo di destinazione non ha chiamato la sua receive. In pratica si è imposto un certo tipo di sincronizzazione tra i processi applicativi;

(2) nella configurazione del nostro sistema può accadere che la sorgente spedisca dati più velocemente di quanti ne possa ricevere il processo destinatario.
Quindi si è pensato di costruire un meccanismo per il controllo di flusso, per permettere la produzione di dati alla velocità del ricevente;

(3) anche se si suppone che il livello di rete fornisca un servizio affidabile, potrebbe comunque accadere che vengono perse alcune unità dati. Per permettere alle entità di trasporto di notificare questa condizione d'errore e di rendere disponibile ai processi applicativi la quantità totale di dati ricevuti o trasmessi, abbiamo scelto di fornire a queste entità un meccanismo per ottenere informazioni riguardanti l'attività al livello di rete.

Queste scelte hanno portato alla definizione di particolari protocolli di rete e di trasporto.
Quando un'a-e chiama la primitiva "send", l'entità di trasporto della stazione chiamante stabilisce una connessione di rete e invia la parte header dell'unità dati di trasporto. Se il processo specificato da "destinazione" non ha ancora chiamato la primitiva "receive" prima dello scadere di un certo timeout, il trasferimento fallisce, la connessione di rete viene rilasciata e la primitiva "send" termina con una eccezione. In caso contrario, quando cioè le entità sono sincronizzate, l'entità di trasporto remota utilizza le informazioni contenute nell'header per verificare la sua disponibilità a ricevere dal partner attuale, nel qual caso manda un messaggio di acknowledge.

L'entità di trasporto sorgente può ora iniziare il trasferimento dei dati dell'area utente nella connessione di rete. L'entità di rete frammenta l'unità dati di trasporto in pacchetti di rete, prima di spedirli; viceversa, quando riceve i caratteri li inserisce direttamente nell'area utente. A livello di rete viene rispedito un ack per ogni messaggio ricevuto correttamente. La perdita di un messaggio di ack informa l'entità di rete mittente dell'esistenza di problemi di comunicazione. In questo caso, tale entità chiude la connessione esistente, e ritorna all'entità di trasporto informazioni riguardanti la quantità di dati spediti, e un codice d'errore. L'entità di trasporto può quindi terminare la primitiva "send", ritornando all'a-e le informazioni ottenute dal livello inferiore. La primitiva "send" termina correttamente quando si è potuto trasferire l'intera unità dati.

Questo meccanismo di acknowledge ha una duplice funzione: (i) permettere il controllo sul trasferimento dati e (ii) sincronizzare i due partner. L'esistenza, a livello di rete, di un'operazione di questo genere, è dovuta a vincoli specifici imposti al nostro sistema di comunicazione.
Infatti, questo meccanismo è stato realizzato per sopperire alla mancanza di informazioni all'interfaccia con il livello link, riguardanti le funzioni fatte sul circuito di link virtuale.

Quando l'entità di rete riceve i pacchetti, toglie loro i caratteri di controllo del protocollo di rete e inserisce i dati direttamente nell'area utente. In questo modo viene evitato un trasferimento inutile in aree temporanee del livello di rete. La primitiva "receive" termina correttamente quando è stata ricevuta tutta la quantità di dati specificata dall'header.

Nel caso in cui un'a-e desideri ricevere dati da un'entità sorgente diversa da quella che ha attualmente iniziato una connessione, non deve far altro che evitare la trasmissione dell'ack, così il trasferimento in corso fallisce e le operazioni di "send" e "receive" terminano con una eccezione.

Se un'entità applicativa non ha predisposto un'area

di memoria sufficientemente grande per contenere l'unità dati spedita dal mittente, l'entità di rete riempie tutto lo spazio disponibile, perdendo tutti i dati in eccesso, dopo di che provocherà la terminazione della "receive" indicando il numero di caratteri effettivamente ricevuti. In questo caso la struttura "csi-exception" contiene la quantità di dati che sono stati persi.

Tra le molte alternative possibili, [9], abbiamo scelto questa per i seguenti motivi; (i) adattarci alle chiamate standard di I/O di questa stazione, (ii) evitare la ritrasmissione di dati che sono già stati ricevuti correttamente, (iii) e perchè non è sempre necessario che l'entità di trasporto debbano rimediare ad errori applicativi. Comunque le due a-e vengono informate della particolare condizione, così possono realizzare un qualche tipo di recovery.

6. NOTE CONCLUSIVE

La realizzazione attuale per i livelli di trasporto e di rete consiste di circa 1300 linee di programma scritte in Pascal. Le funzionalità descritte sono disponibili mediante "function" che devono essere unite al programma applicativo.

L'attività attuale è indirizzata all'ottimizzazione e alla verifica del sistema di comunicazione realizzato. Si pensa, comunque, di realizzare anche un semplice protocollo di file transfer utilizzando i servizi offerti dal sistema di comunicazione.

7. RINGRAZIAMENTI

Desideriamo ringraziare l'ing. A. Mattasoglio e F. Cottini per i loro utili suggerimenti e per il loro aiuto durante la realizzazione.

8. BIBLIOGRAFIA

[1] DATA PROCESSING - OPEN SYSTEM INTERCONNECTION, Basic Reference Model, ISO DP 7498, Dec. 1980

[2] DELTA 16000 PASCAL SYSTEM, Programmer Reference Manual, 1983

[3] VIDEODATA/LAN1, User Reference Manual, 1983

[4] STANDARD ECMA TRANSPORT PROTOCOL, Final Draft, ECMA/TC 24180167, 1980

[5] Rashid R.S., AN INTER-PROCESS COMMUNICATION FACILITY FOR UNIX, Department of Computer Science, Carnegie-Mellon University, April 1981

[6] R.W. Watson, IDENTIFIERS IN DISTRIBUTED SYSTEM, in "Distributed System - Architecture Implementation", Lectures Notes in Computer Science, nr. 105, 1981

[7] LOCAL AREA NETWORK, Logical Link Control, ISO/DP 8802/2, 1983

[8] J.F. Shoch, INTERNETWORKING NAMING, ADDRESSING AND ROUTING, Compcon, 1978

[9] F. Panzieri and B. Randell, INTERFACING UNIX TO DATA COMMUNICATION NETWORKS, Tech. Report 190, the University of Newcastle upon Tyne, 1983

GRAFICA DI BASE PER DLT 1600

Andrea Granelli
Dipartimento di Scienze e Tecnologie Biomediche e Istituto di Cibernetica
Università degli Studi - Milano

RIASSUNTO

Questo articolo descrive l'implementazione di una libreria grafica che fornisce al DLT 16000 alcune capacità grafiche, tra cui il tracciamento di linee, rettangoli e cerchi, lo zoom ed il pan delle immagini, il riempimento di aree rettangolari con diversi patterns ed il tracciamento di caratteri.
Queste capacità grafiche sono gestite da un controller grafico prodotto dalla NEC, IL MDP 7220, collegato al DLT 16000 mediante una interfaccia parallela 8 bit. La parte a più basso livello della libreria grafica (quella che pilota il controller grafico) è scritta in assembler NS 16000, mentre la parte che collega queste procedure al sistema operativo, è scritta in Pascal.

ABSTRACT

This paper describes the implementation of a graphics library which gives DLT 16000 some graphics capabilities, including line, rectangle and circle drawing, zooming and paning of images, box filling with different patterns and character drawing.
Those graphics features are handled by a MPD 7220 graphics display controller manufactured by NEC, which is connected to the DLT 16000 thru a 8-bit parallel interface. The low level part of the library, which monitors the graphics controller, is written in NS 16000 assembler, while the high level part, which links those routines to the operating system environnement, is written in Pascal.

1. CARATTERISTICHE HARDWARE

Il DLT 16000 è dotato di elevate capacità grafiche, essendo collegato, tramite un'interfaccia parallela a 8 bits, con un MPd7220 Graphic Display Controller (GDC), un microprocessore periferico intelligente disegnato per essere il cuore di un sistema grafico raster-scan ad elevate prestazioni.

Collocato tra la memoria grafica (video display memory) ed il bus del microprocessore, il GDC esegue tutte le azioni necessarie per creare l'immagine raster (sincronizzazione del video mediante la generazione dei segnali base di timing, tra cui quello di sync e quello di blanking, conversione parallelo-seriale dei bytes contenenti l'immagine tramite 2 video shift registers) e per gestire la memoria del video.

L'appesantimento nell'elaborazione del computer viene minimizzato dal sofisticato insieme di istruzioni del GDC, comprendente tra l'altro il tracciamento di linee, rettangoli, archi, caratteri e box rettangolari riempiti con un certo pattern ed operazioni come lo zoom e il pan (scroll orizzontale e verticale).
Il GDC si interfaccia alla sua memoria grafica con 20 dei suoi 40 pins (vedi fig. n. 1), assumendosi la totale responsabilità nel controllare tutti gli aspetti di questa interfaccia e quindi generando le sequenze di indirizzi per il raster-scan, gestendo la lettura e la modifica della memoria e la coordinazione di queste attività.

Di questi 20 pins, 16 (AD0..AD15) sono usati per il Data/Address bus bidirezionale, 2 sono di controllo (A16-A17) e gli ultimi 2 (ALE e DBIN/) forniscono le informazioni di timing e di controllo necessarie per coordinare l'hardware esterno con il timing del bus del GDC. Degli altri 20 pins, otto (DO..D7) formano l'interfaccia con l'host computer, mentre i rimanenti servono a svolgere le altre funzioni (sincronizzazione col Timer, input opzionale per la light pen, generazione dei segnali di Blanking e Horizontal sync, Data Bus Input Enable...).

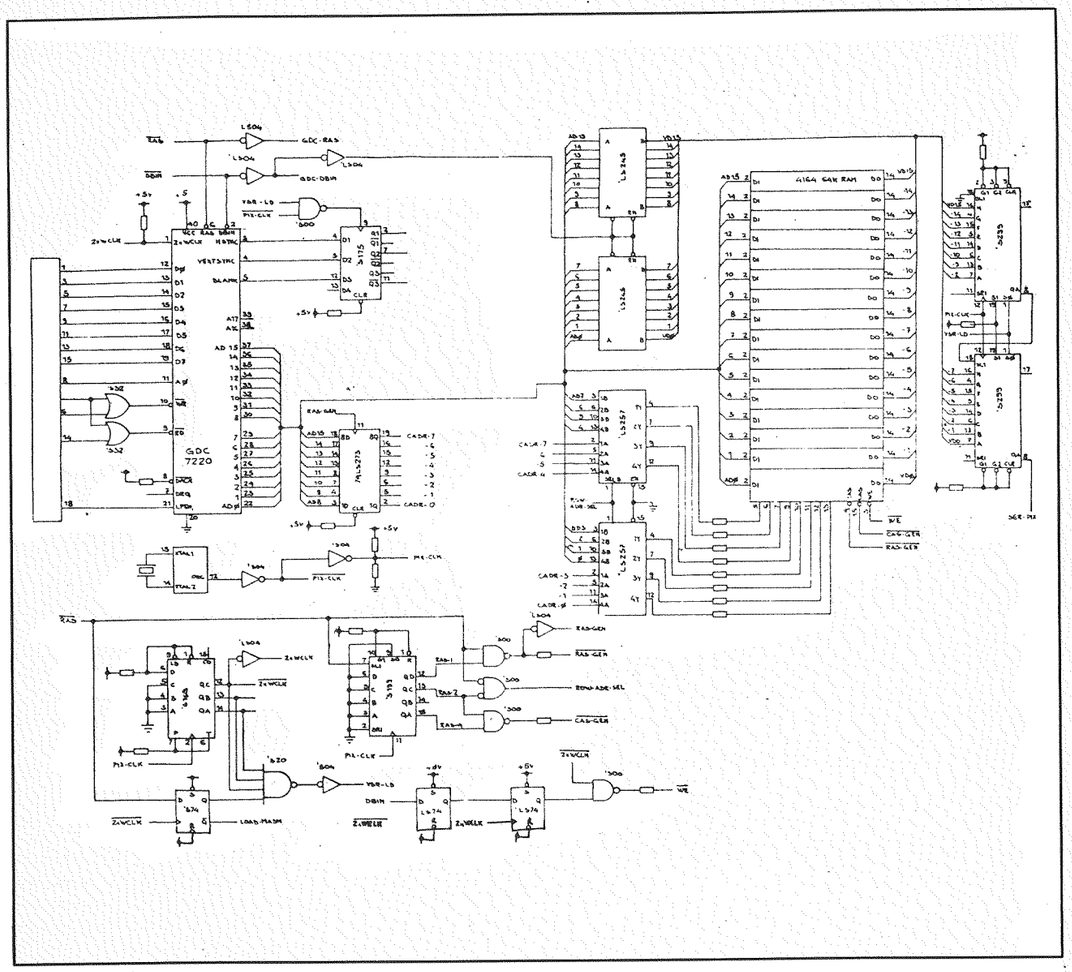


Fig. 1 Schema della piastra grafica

2. IMPLEMENTAZIONE DEL DRIVER SOFTWARE

L'utilizzo di queste caratteristiche grafiche del microprocessore GDC da parte del DLT 16000 è stata resa possibile tramite l'implementazione di una libreria di programmi scritti in Assembler NS 16000 ed in Pascal, che costituisce l'interfaccia software tra il sistema operativo ed il GDC. Questa liberia può essere vista come composta da 3 moduli gerarchizzati.

Quello a più basso livello è composto da un insieme di procedure scritte esclusivamente in Assembler che pilotano il GDC e fanno quindi riferimento agli indirizzi fisici ed alla sintassi dei comandi del microprocessore grafico. Il modulo intermedio, scritto in Pascal, fornisce un livello di astrazione che facilita la creazione di programmi grafici in quanto il suo utilizzo prescinde dalla conoscenza delle caratteristiche del GDC e della sua sintassi di comandi.

Il modulo a più alto livello, invece, anche esso scritto interamente in Pascal, va considerato separatamente in quanto non è una collezione di procedure chiamabili da programma, bensì è un programma eseguibile indipendentemente (è una utility di sistema) che però presuppone l'esistenza delle librerie appena citate. La sua necessità nasce dal fatto che, nella attuale configurazione, non esiste un character generator hardware, per cui la gestione dei caratteri è totalmente software, anche se è in fase di studio l'aggiunta di questo componente (peraltro prevista nella configurazione standard della NEC) sulla piastra.

3. LIVELLO DI BASE

Questo modulo è composto da 11 procedure scritte in Assembler NS 16000, il cui elenco è il seguente

```
PROCEDURE set_gdc;
PROCEDURE clear;
PROCEDURE set cur pos (X,Y, PITCH: WORD);
PROCEDURE read_word (NUM_BYTES: WORD; VAR
                                    ADDR: WORD);
PROCEDURE vect (MODE, X, Y, X1, Y1, PITCH: WORD);
PROCEDURE rect (MODE, WIDTH, HEIGHT, DIR: WORD);
PROCEDURE arco (MODE, RAGGIO, DC, DM, DIR:
                                    WORD);
PROCEDURE fill (MODE, DIR, DC, D2: WORD; VAR
                                    ADDR: WORD);
PROCEDURE style (kind: WORD);
PROCEDURE zoom (HARD, SOFT: WORD);
PROCEDURE scroll (HI_SCREEN, LOW_SCREEN: WORD);
```

"Set_gdc" serve ad inizializzare il sistema grafico mandando al video tutti i segnali di sincronizzazione, blanking ecc...; è variando i parametri presenti in questa procedura che si possono cambiare le caratteristiche del sistema grafico, come per esempio la risoluzione (cambiando il parametro "PITCH", che indica il numero di word per scan-line, attualmente posto a 46) od il passaggio da tracciamento interlacciato a non interlacciato.

"Clear" pulisce lo schermo e "set_cur_pos" setta la posizione corrente (attualmente la risoluzione è O≤=X≤=735 O≤=Y≤=463) mentre la "vect", "rect" e "arco" generano rispettivamente una retta, un rettangolo ed un arco (vedi fig. n. 2). In questo caso è possibile variare anche lo stile od il modo di tracciamento (rispettivamente con il parametro "MODE" o con la procedura "style") o la direzione di tracciamento (parametro "DIR").

La "fill" serve per generare aree rettangolari riempite con un certo pattern ed è la procedura che viene utilizzata anche per il tracciamento di caratteri.
In questo caso il carattere viene definito come una matrice di 8x8 pixels. Su questa procedura agisce la funzione "zoom" che permette di ingrandire il pattern (e la relativa area tracciata) fino a 15 volte, duplicando lungo le assi X e Y, i singoli pixels.

La "read_word", invece, permette di leggere il contenuto della memoria grafica, altrimenti non visibile al DLT 16000 in quanto accessibile solo dal GDC (non è cioè una parte della RAM del DLT 16000). Infine è possibile eseguire degli scroll sia orizzontali (smooth e cioè un pixel alla volta) che verticali (un word alla volta).

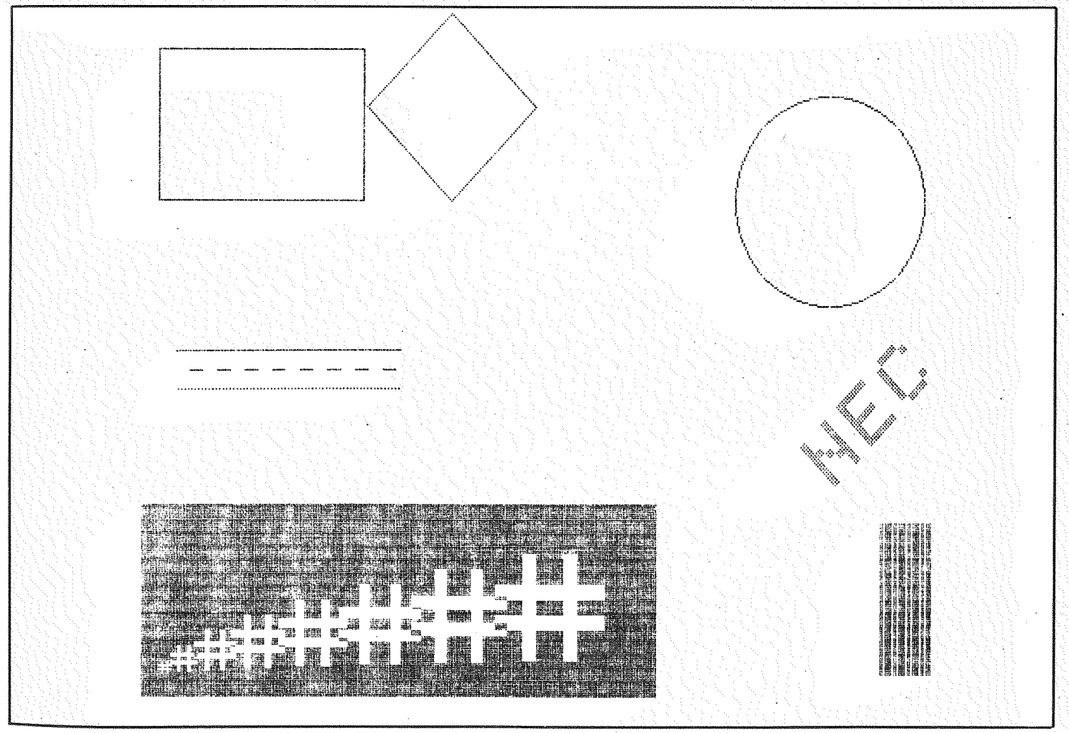


Fig. 2 Funzioni hardware disponibili

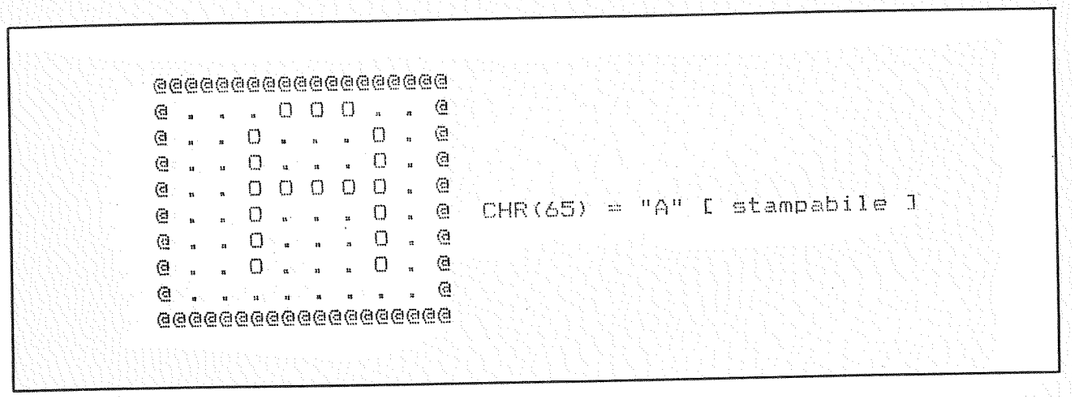


Fig. 3 Menù del CHARACTER GENERATOR

4. LIVELLO INTERMEDIO E INTERFACCIA CON IL PASCAL

Questo modulo è stato implementato mediante una Unit Pascal, chiamata "NEC" ed è il driver vero e proprio; esso è costituito da un insieme di procedure chiamabili da programma Pascal e permette operazioni grafiche più complesse, una maggiore astrazione e fornisce delle funzioni di controllo.

Innanzitutto esegue dei controlli sulle chiamate alle procedure assembler, eliminando possibili casi anomali (filling di box con area 0 o negativa, tracciamento di figure all'esterno del video ecc...); inoltre permette di costruire immagini più complesse (cerchi, gestione separata dei caratteri, complementazione del video) e fornisce una serie di funzioni di controllo molto utili come per esempio la possibilità di memorizzare su di un file porzioni di immagine di dimensione variabile presenti sullo schermo, o trasferite su stampanti grafiche (EPSON FX-80 o SARA11 Honeywell); in questo caso, però, bisognerà utilizzare procedure presenti in un'altra libreria, chiamata "DUMP", che gestisce l'Hardcopy sulle periferiche.

La unit "NEC" contiene inoltre una struttura dati, "graf_instr", che costituisce l'interfaccia tra questo modulo ed il linguaggio grafico CORE, una implementazione dello standard grafico approvato dall'ANSI, che facilita la trasportabilità del software. "Graf_instr" permette di avere una rappresentazione alternativa del disegno correntemente presente sul video.
Un'immagine non viene quindi considerata solamente come un insieme di pixels, ma anche come la collezione di primitive grafiche (move, line, char) necessarie a tracciarla.

5. CHAR GENERATOR SOFTWARE

Questa utility chiamata "CHAR.CODE" permette semplicemente all'utente di gestire via software l'insieme dei caratteri grafici (detto charset). Esso è visto infatti come un PACKED ARRAY [0..255] di Pattern 8x8 memorizzati in un file di nome "SYSTEM.CHARSET". Questo approccio software alla gestione dei caratteri da il grande vantaggio di rendere molto semplice il cambiamento del charset; è infatti sufficiente cambiare il "SYSTEM.CHARSET" con un altro file della stessa struttura ma dal diverso contenuto. L'indice di questo ARRAY corrisponde, nel caso dei caratteri alfanumerici, al loro indice (generato in Pascal dalla funzione "ORD( )") nella tabella ASCII. Il programma vede quindi un carattere come una matrice 8x8 e permette interattivamente di settare o resettare pixels o righe di pixels per creare il carattere desiderato (vedi fig. n. 3).
Può essere inoltre utilizzato per vedere velocemente il contenuto di un certo "SYSTEM.CHARSET".

6. BIBLIOGRAFIA

[1] PRODUCT DESCRIPTION GRAPHICS DISPLAY CONTROLLER mPD 7220, NEC Electronics (Europe), GmbH.
[2] NS 16000 Programmer's Reference Manual.
[3] Sproull, PRINCIPLES OF INTERACTIVE COMPUTER GRAPHICS, Mc Graw Hill Tokio, 1981.
[4] Jensen, Wirth, PASCAL: USER MANUAL AND REPORT, Springer-Verlag, New York, 1978.

IMPLEMENTAZIONE DELLO STANDARD GRAFICO CORE

A. Granelli (*) (%), D. Marini (*), S. Missaglia (*) (%)

RIASSUNTO

Questo articolo descrive l'implementazione dello standard grafico CORE su DLT 16000. In particolare si è previlegiato, per la sua diffusione, l'ambiente raster, su cui si sono concentrati gli sforzi implementativi.

ABSTRACT

This paper describes the implementation of the CORE graphics standard on the DLT 16000 computer. Particulary attention was payed on the raster environment – quite well suited – for which this implementation has been tailored.

1. GLI STANDARD GRAFICI

La riduzione del costo delle memorie centrali degli elaboratori, da un parte, e la sempre crescente diffusione dei personal computer dall'altra, hanno obbligato i produttori di hardware e software grafico ad affrontare e risolvere un problema troppo spesso trascurato: gli standard grafici.

Il problema della trasportabilità del software su altre macchine non si poneva, in quanto ogni casa produttrice aveva il suo mercato più o meno definito e le interferenze dei vari mercati erano piuttosto contenute. Con l'avvento dei personal computer e con la loro capillare diffusione, questo equilibrio di mercati è diventato via via sempre più instabile; come conseguenza il successo di un programma è sempre più legato alla sua possibilità di diffondersi su macchine diverse, e quindi sempre meno legato all'hardware su cui è stato sviluppato; l'attributo di portabilità è diventato una caratteristica imprescindibile del software che voglia essere diffuso.

La portabilità di un programma presuppone da una parte una certa omogeneità nella architettura delle macchine (hardware e software di base) e dall'altra una standardizzazione nella metodologia di programmazione. Proprio per creare questa standardizzazione sono stati proposti e sviluppati linguaggi grafici standard come il CORE SYSTEM elaborato dal GSPC (SIGGRAPH Graphics Standards Planning Committee) ed il GKS (Graphical Kernel System) proposto dalla DIN (Deutches Institut fuer Normung).

Dato che la "portabilità totale" e cioè la trasportabilità dei cosiddetti eseguibili è spesso impraticabile (per una sempre presente disomogeneità hardware), un requisito fondamentale consiste nel minimizzare gli sforzi del programmatore che debba eseguire un determinato programma su di una macchina diversa da quella su cui è stato sviluppato.
In particolare nell'ambito della Computer Graphics accade spesso che il programma in questione "rimanga" sullo stesso computer, ma debba essere adattato per funzionare con periferiche diverse (per esempio video a colori o plotters).

Il desiderio di risolvere questi problemi, ha portato come naturale conseguenza alla riflessione sulla struttura virtuale di un programma, permettendo di definire un importante principio metodologico, che è alla base di ogni tipo di standardizzazione di linguaggi di programmazione e cioè che la parte di programma che fa riferimento ad un particolare dispositivo, deve essere assolutamente isolata dal resto del program-

(*) Istituto di Cibernetica - Università di Milano
(%) Dipartimento di Scienze e Tecnologie Biomediche - Università di Milano.

ma, e quindi facilmente individuabile e modificabile. Questo principio è stato poi formalizzato nel concetto di Driver, che altro non è che una collezione di sottoprogrammi, scritti spesso in linguaggio macchina, che gestiscono la comunicazione tra il computer e le varie periferiche – in questo caso i dispositivi grafici.
L'identificazione poi di un insieme limitato di funzioni base, dette primitive grafiche in quanto essenziali per la realizzazione di un qualsiasi programma grafico, costituisce l'altra caratteristica di un linguaggio standard, e cioè la completezza.

Il nostro lavoro ha fatto riferimento al linguaggio CORE in quanto nel 1982, anno in cui è incominciata l'implementazione, era sicuramente lo standard più diffuso. Inoltre il GKS (l'altro standard grafico di una certa importanza [vedi Note di Software 23/24] pressuppone la creazione di una grossa struttura dati (Workstation-State-List), Workstation-Description-Table, GKS-State_List, ...) che lo rende difficilmente utilizzabile su personal computer (data la loro scarsa disponibilità di memoria centrale).
Diamo ora un breve resoconto sulle caratteristiche principali del CORE, specificando in particolare le funzioni implementate.

2. I LIVELLI DEL CORE

Esistono una molteplicità di periferiche aventi prerogative a volte molto differenziate; i propositori dello standard hanno provveduto a fornire delle classi di livelli che evitino la nascita e la proliferazione di "dialetti" di volta in volta comprendenti questa o quella funzionalità.'

Per dialetto si intende la introduzione in un certo linguaggio di funzioni caratteristiche di una particolare configurazione hardware/software. Questa operazione rende il nuovo linguaggio una "variante dialettale" del precedente, in quanto esso non è più "interamente trasportabile". Da qui l'introduzione dei livelli compatibili verso l'alto, cioè tali per cui un programma scritto utilizzando un certo livello del CORE (per es. 1-1-2D) sia sempre trasportabile su computers provvisti di un CORE di livello uguale o superiore (per es. 3-1-3D).

Le tre classi di livelli compatibili verso l'alto sono:

Output, Input, Dimensione.

I livelli per le funzioni di output del CORE sono:

1) BASE primitive, attributi delle primitive, visualizzazione e controllo, segmenti temporanei.
2) BUFFERIZZATO segmenti ritenuti, attributi di segmento.
3) DINAMICO attributi di trasformazione, attributi di sensibilità.

I livelli per le funzioni in Input sono:

1) NULLO nessuna funzione di Input.
2) SINCRONO inizializzazione della periferica, interazione sincrona, controllo dell'echo.
3) COMPLETO abilitazione e disabilitazione esplicita, controllo delle code di eventi, campionatura, associazioni.

I livelli per la dimensionalità del CORE sono:

1) 2D tutte le funzioni sono nel piano
2) 3D tutte le funzioni sono nello spazio e sono visualizzati gli oggetti contenuti in un tronco di piramide.

L'attuale implementazione raggiunge il livello 2 Output, 1 Input, 1 Dimensione.

3. ELENCO DELLE FUNZIONALITÀ IMPLEMENTATE

Dopo una approfondita analisi del linguaggio CORE si è deciso di non implementare tutte le funzionalità, visto la inutilità di alcune in un ambiente basato su terminali raster. Ci riferiamo in particolare a tutte le funzioni relative alla gestione dei terminali a persistenza, come il DVST della Tektronix, caratterizzati dalla impossibilità di cancellare una parte dell'immagine senza cancellare tutta l'immagine. Poichè invece il CORE prevede questa operazione (quando si modifica l'attributo di visibilità di un segmento), gli ideatori del CORE hanno pensato di permettere all'utente di cancellare virtualmente una parte dell'immagine e rimandare l'aggiornamento del terminale fino a che non venga richiesto esplicitamente. In questo modo è possibile eseguire più cancellazioni logiche ma una sola cancellazione fisica.

Fatte queste considerazioni, descriviamo brevemente l'implementazione, non entrando nel dettaglio delle specifiche del linguaggio CORE, per le quali si rimanda alla bibliografia allegata, ma elencando, opportunamente commentate, le funzionalità implementate. Possiamo, seguendo le specifiche del CORE, suddividerle in 4 classi.

GESTIONE DELLA PERIFERICA VIRTUALE

PRIMITIVE DI OUTPUT

SEGMENTAZIONE

ATTRIBUTI

Gestione della periferica virtuale

Il CORE garantisce l'indipendenza del software grafico delle periferiche utilizzate, mediante il concetto

di periferica virtuale. In pratica viene introdotto un nuovo sistema di coordinate, detto NDC (Normalized Device Coordinate) tra le coordinate utente (dette WC cioè World Coordinates o DC) (vedi figura 1).
Vi sono quindi due funzioni che permettono i legami tra i 3 sistemi di coordinate:

Set_window, Set_Vieport

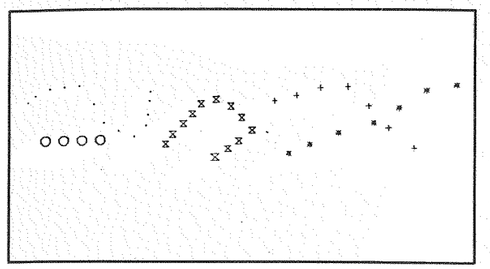


Fig. 1 I tre sistemi di coordinate

3.1 Primitive di output

Gli oggetti grafici del mondo utente sono descritti al sistema invocando queste primitive, che sono affette dai valori correnti dei loro attributi (posizione corrente, stile di tracciamento ecc...) e che vengono specificate mediante le coordinate utente

move_rel_2, move_abs_2,
line_rel_2, line_abs_2,
marker_rel_2, marker_abs_2,
poly_l_rel_2, poly_l_abs_2, (sono le POLYLINE)
poly_m_rel_2, poly_m_abs_2, (sono le POLYMARKERS)
text_2
escape

3.2 Segmentazione

Il CORE fornisce la possibilità di strutturare un'immagine, di vederla cioè composta da sottoimmagini, dette segmenti, dotate di alcuni attributi, come per esempio la visibilità. Questo permette di poter eseguire delle operazioni solo su alcune parti dell'immagine, senza coinvolgere tutta la struttura di rappresentazione. Per fare ciò si utilizza il concetto di segmento che altro non è che una collezione di primitive di output con un nome. Nel CORE sono previsti due tipi di segmenti: quelli temporanei, non rivisualizzabili, e quelli ritenuti.
La gestione dei segmenti avviene con le seguenti funzioni:

create_temp_seg, close_temp_seg, inq_open_temp_seg
create_retain_seg, close_retain_seg, inq_open_retain_seg
rename_retain_seg, delete_retain_seg, delete_all_retain_seg

3.3 Attributi

Gli attributi servono a definire le caratteristiche delle singole primitive di output e dei segmenti; ogni oggetto, primitiva o segmento, è creato secondo i valori correnti, mentre possono essere cambiati successivamente gli attributi dei segmenti ritenuti. Parleremo di attributi statici quando non possono essere più modificati, e dinamici se invece possono esserlo.

Vi sono due classi di funzioni che si applicano agli attributi: quelle che modificano il valore dell'attributo (il cui suffisso è Set), e quelle che ritornano il valore dell'attrito corrente (il cui suffisso è Inquire).

STATICI
Set_color, Set_marker, Set_Line_width, Set_line_style

DINAMICI
Set_visibility, Set_hilight

INQUIRE

In figura 2 vi è un esempio dell'utilizzo dell'attrito "marker-symbol", con valori differenti.

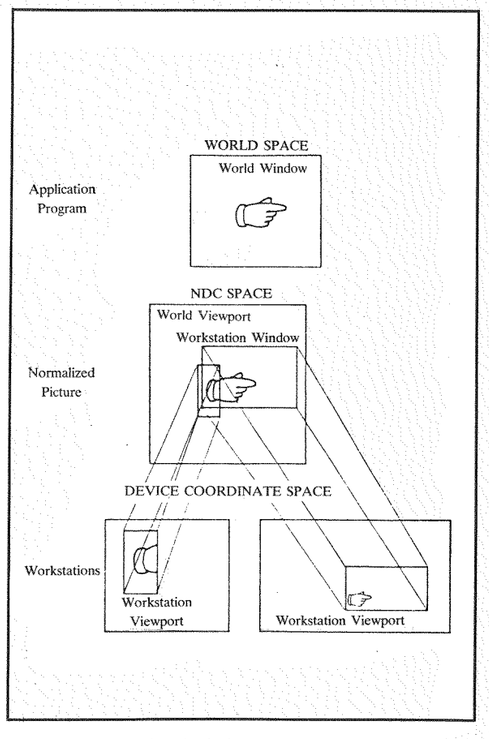


Fig. 2 Differenti valori dell'attributo marker_style

4. CONCLUSIONE

Si è scelto di implementare un sottoinsieme del livello 2b senza input del CORE che trovasse un compromesso tra funzioni disponibili all'utente e memoria occupata; si potrebbe pensare che, procedendo in questo modo, si vada contro la logica stessa della standardizzazione dei linguaggi che vieta, per sua definizione, anche piccole deviazioni delle specifiche definite, perchè ciò potrebbe far nascere dei dialetti; ma ciò non è così, infatti questa implementazione non fornisce strutture addizionali, ma semplicemente evita la presenza di funzioni che rimarrebbero inutilizzate; questo perchè il CORE, essendo un linguaggio standard grafico, deve permettere la portabilità del software su qualsiasi macchina e quindi, per esempio, anche su i vecchi terminali DVST (Direct View Storage Terminal) e per questo motivo deve contenere istruzioni anche per questo tipo di terminale. (Begin-Batch-of-Update, ecc...).

Nel nostro caso, essendo l'ambiente definito a priori, si è preferito perdere un po' di generalità (implementando cioè il CORE per sistemi Raster-graphics, presenti sulla maggior parte del personal computer), ma rendere l'oggetto "utilizzabile praticamente" (la libreria che contiene le procedure del CORE e i Drivers occupa infatti circa 15K bytes).

Lo scopo finale era creare un ambiente didattico dove si potesse imparare ad usare un linguaggio standard grafico; quindi non sono state implementate tutte le funzioni prescritte dal livello 2b senza input.

Essendo il DLT 16000 dotato di un sistema grafico ad elevate prestazioni (tracciamento hardware di cerchi, rettangoli, riempimento di aree ecc.) basato su un Graphic Display Controller della NEC, il uP 7220, si è pensato di utilizzare queste caratteristiche da CORE e per fare ciò si è utilizzata la funzione ESCAPE, prevista dallo standard, il cui scopo è proprio quello di "utilizzare in maniera standard caratteristiche non standard" della periferica utilizzata.

È chiaro quindi che la presenza, nella proposta CORE di una funzione come ESCAPE, mette in luce una importante caratteristica: l'esistenza di uno standard reale è spesso una chimera; esso deve quindi essere visto piuttosto come una mediazione fra due caratteristiche ugualmente importanti. Da una parte l'utilizzo di una metodologia di programmazione sufficientemente standard da permettere facilmente il trasporto di programmi e programmatori, e dall'altra il pieno utilizzo delle caratteristiche hardware di un sistema grafico.

5. BIBLIOGRAFIA

[1] STATUS REPORT OF THE GRAPHIC STANDARD PLANNING COMMITEE, Computer Graphics, a quarterly Report of Siggraph-ACM, vol. 13, N. 3, 1979

[2] COMPUTER GRAPHICS, CAD, ELABORAZIONE DELLE IMMAGINI: SISTEMI E APPLICAZIONI, Atti del 3 Convegno Nazionale della AICOGRAPHICS, Milano 1983

[3] Newman, Sproull, PRINCIPLES OF INTERACTIVE COMPUTER GRAPHICS, Mc Graw Hill, Tokio, 1981

[4] Jensen, Wirth, PASCAL: USER MANUAL AND REPORT, Springer-Verlag, New York, 1978

ALGORITMI DI ANALISI PER RETI DI PETRI

Daniele Pieragostini
Istituto di Cibernetica - Università di Milano

RIASSUNTO

Il trattamento di sistemi sempre più complessi richiede di necessità un approccio basato su adeguati formalismi, ed in particolar modo su strumenti automatici di analisi. In questo articolo viene descritto un tale strumento, basato sulla teoria delle reti di Petri, che consente, a partire dall'analisi di particolari proprietà delle reti, di stabilire la correttezza funzionale e strutturale del modello.

ABSTRACT

Dealing with more and more complex systems, a formal design methodology supported by automatic tools is required. This paper describes a software package for verifing and validating models described with Petri Nets. The analisys consists on checking different kinds of net properties from which inferring the behaviour of the described system.

1. INTRODUZIONE

La descrizione di sistemi (tecnologici e non) sempre più complessi comporta una altrettanto crescente difficoltà nell'analisi e nella verifica delle loro proprietà. Per questo motivo sono necessari strumenti formali sempre più sofisticati in grado di descriverne le componenti peculiari e fornire informazioni sul loro funzionamento.

Fra questi formalismi quello che sembra avere buone possibilità di imporsi come lo "standard" dei modelli di specifica è la teoria delle reti di Petri. Storicamente le reti di Petri nascono all'inizio degli anni '60 come uno strumento semplice e naturale, ma al contempo rigoroso, atto a modellare sistemi in cui occorra considerare flussi ordinati di informazione.

Durante questi anni lo studio delle reti di Petri ha avuto un notevole sviluppo principalmente in ambito universitario, pertanto in una prospettiva sostanzialmente teorica.
Ultimamente però si sta ponendo l'accento anche sull'aspetto applicativo delle reti di Petri; è sintomatico in questo senso l'interesse che questa metodologia sta riscontrando da parte di chi, operando in certi settori industriali, si occupa in particolare della descrizione e dell'analisi di problematiche hardware e software.

Tuttavia la mancanza di "tools" adeguati allo studio e alla manipolazione delle reti ne ha probabilmente limitato la diffusione, dato che l'analisi "manuale" di una rete, anche di modeste dimensioni, risulta assai complessa, e quindi sostanzialmente impraticabile.

Per questo motivo è stato sviluppato uno strumento di analisi delle proprietà matematiche delle reti che, avvalendosi dell'ausilio del computer, consente di affrontare la modellazione sistemica in modo semplice ed efficace.
Queto package fa parte di un più ampio progetto relativo alle reti di Petri tuttora in fase di sviluppo presso l'Istituto di Cibernetica dell'Università di Milano, che comprende, tra l'altro, un editor grafico per reti [2], la cui completa integrazione con i programmi di analisi rende l'insieme uno strumento sofware di notevoli possibilità.

In questa nota vengono descritti gli algoritmi che compongono questo pacchetto applicativo, per quel che riguarda le loro principali caratteristiche strutturali e funzionali, dopo una rapida e informale panoramica sulle proprietà matematiche che sono alla base dell'implementazione degli algoritmi. Nell'esposizione si assume che il lettore abbia una certa familiarità con i concetti basilari della teoria delle reti di Petri. In ogni caso, per una più approfondita trattazione sui vari aspetti della teoria delle reti di Petri rimandiamo il lettore alle ormai numerose pubblicazioni (ed in particolare a [5]).
L'ultima parte dell'articolo è dedicata all'illustrazione di un esempio di applicazione.

2. LE RETI DI PETRI

Le reti di Petri forniscono un valido strumento di

specifica e di analisi di sistemi, in particolare sistemi discreti che modellino flussi di dati, interazione di processi, accesso ordinato a risorse etc. Quando il modello che del sistema si è scelto viene tradotto in una rete, occorre verificarne la correttezza e la consistenza rispetto alle ipotesi iniziali; tale verifica avviene attraverso l'analisi delle proprietà matematiche della rete stessa, dalla cui conoscenza è possibile derivare precise informazioni sul comportamento del sistema modellato.

3. PROPRIETÀ DELLE RETI DI PETRI

Molto schematicamente possiamo suddividere le proprietà delle reti in due classi: le proprietà dinamiche, che riguardano l'evoluzione della rete, e dipendono quindi dalla marcatura iniziale (cioè dallo stato iniziale) della rete (fig. 1); le proprietà statiche, che derivano unicamente dalla struttura della rete, indipendentemente dal tipo di evoluzione a cui questa sarà soggetta.

3.1. Proprietà dinamiche

Le proprietà dinamiche sono le seguenti:

- vivezza
- ciclicità
- sicurezza
- resettabilità
- limitatezza

Chiariamo questi concetti ricorrendo ad alcuni esempi.

Si dice che una rete è "VIVA" se, a partire da una qualsiasi marcatura e per ogni transizione della rete, esiste un cammino che abilita allo scatto quella transizione. La non vivezza di una rete implica la presenza di un deadlock (blocco dell'evoluzione della rete) o di una sotto-rete non attiva; una rete non viva corrisponde alla presenza di un blocco nel funzionamento del sistema modellato.

Una rete è "CICLICA" se tutte le sue marcature (i suoi stati) sono riproducibili; nel caso in cui l'insieme delle marcature riproducibili sia un sottoinsieme proprio dell'insieme delle marcature possibili della rete, e comprenda quella iniziale, si dice che la rete è "RESETTABILE". Questa proprietà si traduce evidentemente in un analogo comportamento da parte del sistema modellato (indicando in sostanza la possibilità di ritornare in un qualsiasi stato precedentemente ottenuto).

Una rete si dice "SICURA" se durante l'evoluzione non hanno luogo situazioni in cui una transizione non può scattare perchè un sottoinsieme dei suoi posti di uscita è marcato (si dice in questo caso che ha luogo

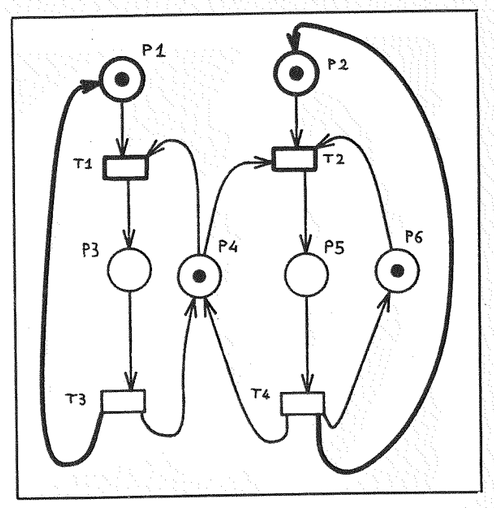


Fig. 1

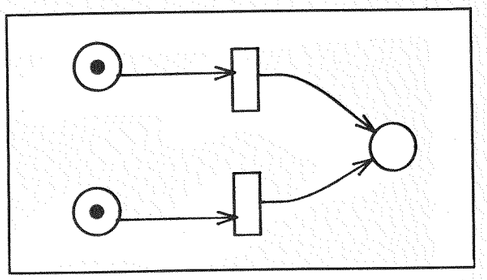


Fig. 2

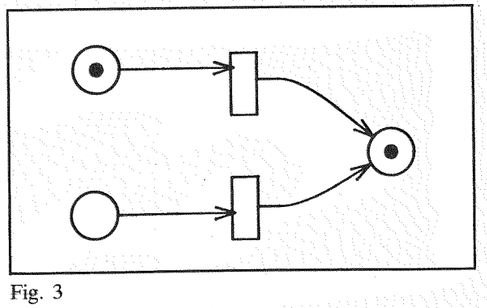


Fig. 3

una situazione di "contatto"). La nozione di sicurezza è nata essenzialmente per descrivere, in reti CE, situazioni anomale, come ad esempio quella rappresentata in figura 2; qui le due transizioni non possono scattare contemporaneamente perchè il posto in uscita non può contenere più di una marca. L'evoluzione della rete può portare solo ad una situazione del tipo di quella illustrata in figura 3, che rappresenta per l'appunto un contatto.

Il passo successivo è stato quello di generalizzare la nozione di sicurezza imponendo l'assenza di contatti durante tutta l'evoluzione della rete, sottintendendo con questo il fatto che un eventuale contatto possa derivare da situazioni anomale come quella appena descritta.

Una rete si dice "LIMITATA" se il numero di marche all'interno dei posti non supera un numero K fissato (il che significa che nessun posto può ospitare un numero illimitato di marche). Questa proprietà entra in gioco unicamente per reti PT, in cui la capacità dei posti può essere arbitrariamente grande (anche infinita); viceversa, una rete CE è sempre limitata per definizione. Una rete non limitata evidentemente descrive una entità che normalmente non ha riscontro fisico, in quanto generalmente si ha a che fare con sistemi finiti che gestiscono risorse in numero finito.

3.2. Proprietà statiche

Le proprietà statiche si possono ridurre ai concetti di invarianti di posti e di invarianti di transizioni.

Un P-INVARIANTE è un sistema di posti della rete tale che la somma delle marche contenute nei posti medesimi non varia durante l'evoluzione della rete. Viceversa un T-INVARIANTE è un insieme di transizioni il cui scatto complessivo non cambia la marcatura della rete (o meglio data una marcatura M, la sequenza di scatti riporta la rete esattamente in M).

Dall'analisi degli invarianti è a volte possibile dedurre alcune informazioni sul comportamento dinamico della rete.
Esistono infatti delle condizioni o solo necessarie o solo sufficienti che permettono, in particolari casi, di decidere della validità di particolari proprietà dinamiche.

4. CARATTERISTICHE GENERALI DELL'IMPLEMENTAZIONE

I programmi sono stati sviluppati sul DLT 16000 che supporta una versione non standard del linguaggio Pascal (precisamente l'UCSD Pascal). Durante la fase di scrittura dei programmi si è fatto uso unicamente di procedure del Pascal standard, o, quantomeno, di "features" facilmente simulabili sotto altri compilatori Pascal; tutto ciò al fine di ottenere per i medesimi la massima portabilità possibile (attualmente ne esistono versioni per PC IBM e compatibili, e per HP 9836). I singoli codici occupano mediamente 20 Kbyte di memoria, per un totale di circa 120 Kbyte. Trattandosi di algoritmi complessi ed estremamente dispendiosi (ciò che deriva da una intrinseca complessità nelle tecniche di analisi delle reti), si è cercato di contenere entro limiti ragionevoli sia i tempi di esecuzione che l'occupazione di memoria run-time.

Il risultato ottenuto, che riteniamo soddisfacente tenuto conto delle difficoltà che il problema offre, è tale da rendere possibile l'analisi di reti compatibilmente con la velocità di elaborazione e con il limite fisico della memoria centrale.

L'unico vincolo imposto riguarda le dimensioni delle reti: non è possibile infatti analizzare reti con più di 256 posti e 256 transizioni. Tale limite è in ogni caso del tutto teorico, in quando difficilmente si potrebbero concepire reti di tali dimensioni (che comunque risulterebbero assolutamente ingestibili).

Per quanto riguarda le prestazioni dei programmi, è difficile dare indicazioni precise, dipendendo queste oltrechè dalle dimensioni della rete, anche dalla complessità della medesima (numero e tipo di configurazione degli archi uscenti dai nodi), dal livello di parallelismo, dalla marcatura iniziale etc. In ogni caso, per dare delle indicazioni di massima, diremo che per reti di "ragionevole complessità" valgono, per i tempi di esecuzione, i seguenti ordini di grandezza:

reti 10 posti x 10 transizioni: qualche secondo
reti 20 posti x 20 transizioni: decine di secondi
reti 30 posti x 30 transizioni: qualche minuto

4.1 Struttura dati

Le reti di Petri vengono usualmente descritte tramite matrici NxM (con N numero dei posti ed M numero delle transizioni della rete) dette "matrici di incidenza" (figura 4); gli elementi di queste matrici sono:

M(i,j) = 0 se il posto i-esimo e la transizione j-esima non sono collegati
M(i,j) = –1 se l'arco va dal posto i-esimo alla transizione j-esima
M(i,j) = 1 se l'arco va dalla transizione j-esima al posto i-esimo

	T1	T2	T3	T4
P1	–1	0	1	0
P2	0	–1	0	1
P3	1	0	–1	0
P4	–1	–1	1	1
P5	0	1	0	–1
P6	0	–1	0	1

Fig. 4 - Matrice di incidenza della rete di figura 1.

Una soluzione alternativa è quella di usare una struttura ricorsiva, in cui ogni nodo è un record del tipo;

```
nodo = RECORD
          tipo_nodo : (posto, transizione);
          successori : lista_nodi;
          predecessori : lista_nodi;
          next: ^nodo
       END;
lista_nodi = RECORD
                oggetto : ^nodo;
                next : ^lista_nodi
             END;
```

Per motivi di efficienza e di comodità è stata adottata la rappresentazione matriciale. Per descrivere una matrice abbiamo evitato sia l'uso di array (che vanno dimensionati prima della compilazione e quindi prima di conoscere le dimensioni della rete) sia l'uso di liste (efficienti dal punto di vista del consumo della memoria, ma onerose per quanto riguarda i tempi di accesso). Per ottenere un più alto livello di astrazione e quindi una maggiore chiarezza nel codice, si è fatto uso di un ADT (Abstract Data Type) basato sulla seguente struttura base:

```
TYPE
      mat = array [1..16,1..16] of integer;
      punt_mat = ^mat;
      matrice = array [1..16,1..16] of punt_mat;

      vet = array [1..16] of integer;
      punt_vet = ^mat;
      vettore = array [1..16] of punt_vet;
```

cioè matrici (vettori) di puntatori che puntano a matrici (vettori) di interi. L'uso di questa struttura dati unisce i vantaggi di una struttura dinamica alla facilità di accesso di una struttura statica.
Chiaramente in questo modo si limita la dimensione massima delle reti a 256x256 (limite che supera di molto le reali possibilità di analisi di una rete), ma al contempo si limita al massimo lo spreco di memoria dovuto alla gestione di matrici le cui dimensioni non siano multiple di 16.
L'ADT è costituito dal seguente insieme di funzioni e procedure per l'accesso e la manipolazione degli elementi della matrice

```
CREATE_MATRIX (VAR nome : matrice; num-
                righe, num_colonne : integer)
GET_ELEMENT (nome : matrice; riga, colonna :
                integer) : integer
DELETE_COLUMN (VAR nome : matrice; colon-
                na: integer)
ADD_ROW (VAR nome : matrice; riga : vettore)
ADD_COLUMN (VAR nome : matrice; colonna :
                integer)
```

5. GLI ALGORITMI

Gli algoritmi di analisi realizzano globalmente le seguenti funzioni:

- Costruzione dell'insieme delle marcature raggiungibili
- Costruzione del grafo delle marcature
- Verifica delle seguenti proprietà:
  - Vivezza
  - Ciclicità
  - Sicurezza
  - Presenza di deadlock
  - Resettabilità
  - Limitatezza
  - Conservazione della marcatura
- Calcolo degli invarianti di posti e di transizioni
- Riduzione di reti
- Simulazione dell'evoluzione dinamica della rete

I dati di ingresso (cioè la rete da analizzare) vengono inseriti graficamente tramite l'apposito EDITOR (cfr. [2]) integrato con i programmi di analisi. Esiste però anche una versione dei medesimi totalmente indipendente dall'editor grafico in cui la rete viene inserita tramite la usuale notazione matriciale (cfr. [5]).

L'uso di questi programmi non comporta alcuna difficoltà per l'utente: le uniche conoscenze richieste riguardano ovviamente le nozioni sulla teoria delle reti di Petri; il testo che ha fornito il necessario supporto teorico per l'implementazione algoritmica è "Petrinetze" di W. Reisig.

Descriviamo ora quella che potrebbe considerare una tipica sessione di analisi.

Dopo aver disegnato la rete tramite il Net-editor (non avendo a disposizione quest'ultimo è consigliabile inserire i dati della rete – matrici di flusso, marcatura inziale, capacità dei posti – in un file che poi verrà ridiretto come input per i programmi di analisi), l'utente ha quattro possibilità;

- iniziare direttamente con l'analisi dinamica della rete (costruzione dell'insieme delle marcature raggiungibili, del grafo delle marcature e verifica delle proprietà);
- effettuare l'analisi strutturale (calcolo degli invarianti);
- procedere ad una preventiva riduzione della rete in modo da facilitare le successive operazioni. L'algoritmo di riduzione trasforma la rete originaria in una rete con un minor numero di posti e di transizioni, ma avente le stesse proprietà di quella di partenza.
- effettuare subito una verifica diretta del comportamento dinamico della rete utilizzando il programma di simulazione.

Chiaramente queste operazioni non sono mutuamente esclusive; anzi è spesso utile effettuarle tutte al fine

di ottenere il maggior numero di informazioni sul comportamento della rete analizzata.

Al fine di ottimizzare i tempi di analisi, l'algoritmo che costruisce il grafo delle marcature individua subito un eventuale errore nella costruzione della rete (traducibile nella presenza di un contatto o di un deadblock) ed emette un apposito messaggio: a questo punto l'utente può decidere di continuare o interrompere l'analisi. In quest'ultimo caso, individuato l'errore, può richiamare il Net-editor e modificare la rete per poi procedere ad una successiva analisi.

6. UN ESEMPIO DI APPLICAZIONE

Vediamo ora con un esempio la modellazione e la relativa analisi di un sistemaa descritto con una rete di Petri. Allo scopo abbiamo scelto un protocollo di trasmissione RS232. Si tratta in sostanza di rappresentare l'insieme di attività che si succedono all'interno dell'interfaccia tra un DCE e un DTE prima, durante e dopo la trasmissione, in particolare per quel che riguarda il comportamento del protocollo a fronte di eventuali cadute dei segnali.
I segnali presi in considerazione sono solo una parte dei ventuno previsti dallo standard RS232, vale a dire quelli correntemente utilizzati nella maggioranza delle implementazioni.

La rete ottenuta, che è di tipo CE, è illustrata in figura 5. La corrispondenza tra nomi dei posti e delle transizioni viene fornita nella tabella 1.
La marcatura iniziale (posti 1, 3 e 4) corrisponde ad

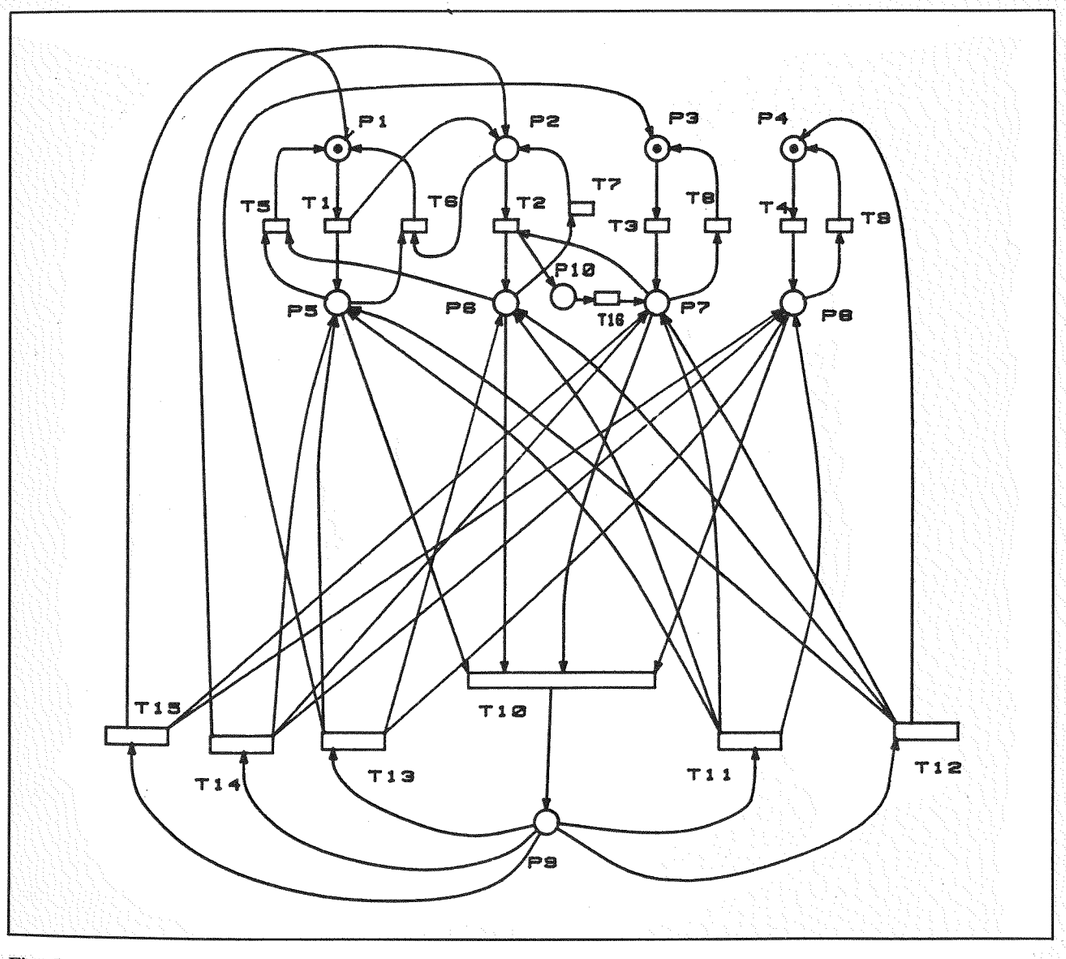


Fig. 5

una situazione nella quale tutti i segnali sono disattivati.
L'unico vincolo temporale imposto al modello consiste nel fatto che il DSR del DCE e l'RTS del DTE vengano attivati prima del segnale di CTS del DCE, e ciò è evidenziato dalla disposizione delle transizioni T1 T2 e T3. Il posto P9 rappresenta la condizione di trasmissione in atto (tutti i segnali sono attivi).

Le transizioni T5 T6 T7 T8 e T9 rappresentano eventuali cadute dei rispettivi segnali prima dell'inizio della trasmissione. Le transizioni T12 T13 T14 e T15 rappresentano viceversa cadute dei segnali durante la fase di trasmissione con conseguente rinizializzazione; la transizione T11 rappresenta la caduta della trasmissione dovuta sia alla corretta terminazione della trasmissione stessa che ad una interruzione provocata dall'utente o da cause esterne.
Il modello rappresentato segue, almeno per i segnali considerati, le indicazioni della raccomandazione CCITT V.24.

In tabella 2 è riportato l'insieme di tutte le possibili marcature della rete descritto come sequenza di vettori a 9 componenti: gli uni e gli zeri indicano rispettivamente la presenza e l'assenza della marca nei posti relativi.
In tabella 3 è riportato il grafo della marcature in forma chiaramente alfanumerica: si tratta di una lista di terne di numeri interi che specificano rispettivamente il nodo di partenza (marcatura i-esima) l'arco (la transizione j-esima) e il nodo di arrivo (marcatura i+esime). I messaggi che seguono corrispondono ad altrettante proprietà della rete. La tabella 4 riporta gli invarianti di posti e di transizioni, rappresentati come vettori a p o a t componenti.

I 4 P-invarianti individuano gruppi di posti la cui marcatura complessiva (uguale a 1 in questo caso) resta costante durante l'evoluzione della rete. Considerando ad esempio il primo invariante (tabella 4a), vediamo che esso è formato dai posti P1 P5 e P9; deduciamo da ciò che durante l'evoluzione della rete solo uno di questi può essere marcato in quanto la somma complessiva delle marche è costante e uguale a 1; di conseguenza la rete non potrà mai trovarsi in più di uno di questi stati contemporaneamente.
I 10 T-invarianti (tabella 4 b) corrispondono ad altrettante sequenze cicliche di scatto all'interno della rete (e rappresentano altrettanti processi di rinizializzazione).

Concludendo, possiamo dire che la rete possiede tutte le "buone proprietà" statiche e dinamiche, e che quindi il modello risponde, sia da un punto di vista strutturale che da un punto di vista funzionale, alle ipotesi fatte in partenza.
Da notare che la rete così come è stata costruita si presenta nella sua forma più essenziale, pertanto non è possibile ottenerne una ridotta.

Da questa esposizione abbiamo escluso l'algoritmo di simulazione in quanto, trattandosi di un programma essenzialmente interattivo, non si presta ad essere descritto in questa sede.

7. CONCLUSIONE

È stato presentato uno strumento software per la verifica delle proprietà matematiche delle reti di Petri, tramite il quale è possibile avere precise indicazioni sul livello di affidabilità dei modelli sistemici. Con un esempio ne abbiamo descritto gli aspetti più interessanti e mostrato le potenzialità.
Attualmente si sta valutando la possibilità di estenderne la validità a tipi di reti più complessi.

8. RINGRAZIAMENTI

L'autore ringrazia il Prof. Gianni Degli Antoni per il costante incoraggiamento e i numerosi spunti che hanno accompagnato la realizzazione di questo lavoro.
Un ringraziamento particolare va anche a Tiziana Campanella, Roberta Fossati, Carla Garavaglia, ed Elena Orazi che hanno ideato la rete che modella il protocollo RS232.

9. BIBLIOGRAFIA

[1] B. Chezalviel-Pradin, UN OUTIL GRAPHIQUE INTERACTIF POUR LA VERIFICATION DES SYSTEMES A EVOLUTION PARALLELE DECRITS PAR RESEAUX DE PETRI, These de decteur ingenieur, Université Paul Sabatier, Toulouse, 1979

[2] M. Negri, E. Ciapessoni, UN EDITOR PER RETI DI PETRI IN UNA METODOLOGIA DI COSTRUZIONE DI SISTEMI COMPLESSI, Note di Software n. 23/24, 1984

[3] J. Peterson, PETRI NET THEORY AND THE MODELLING OF SYSTEMS, Prentice-Hall, 1981

[4] D. Pieragostini, PROGETTAZIONE DI UN LABORATORIO DI RETI DI PETRI, Tesi di Laurea, Istituto di Cibernetica, Università di Milano, 1984

[5] W. Reisig, LE RETI DI PETRI, Mondadori, 1984

P1 = not (RTS DTE) and not (CTS DCE)
P2 = (RTS dce) and not (CTS dce)
P3 = not (DSR dce)
P4 = not (DTR dte)
P5 = (RTS dte) and not (TD dte) and not (RD dce)
P6 = (CTS dce) and not (TD dte) and not (RD dce)
P7 = (DRS dce) and not (TD dte) and not (RD dce)
P8 = (DTR dte) and not (TD dte) and not (RD dce)
P9 = (RTS dte) and (CTS dce) and (TD dte) and (RD dce) and (DRS dce) and (DTR dte)
P10 Nessun valore

Tabella 1 Espressioni logiche associate ai posti della rete

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
1	1	0	1	1	0	0	0	0	0	0
2	0	1	1	1	1	0	0	0	0	0
3	1	0	0	1	0	0	1	0	0	0
4	1	0	1	0	0	0	0	1	0	0
5	0	1	0	1	1	0	1	0	0	0
6	0	1	1	0	1	0	0	1	0	0
7	1	0	0	0	0	0	1	1	0	0
8	0	0	0	1	1	1	0	0	0	1
9	0	1	0	0	1	0	1	1	0	0
10	0	0	0	0	1	1	0	1	0	1
11	1	0	0	1	0	0	0	0	0	1
12	0	1	0	1	1	0	0	0	0	1
13	0	0	0	1	1	1	1	0	0	0
14	1	0	0	0	0	0	0	1	0	1
15	0	1	0	0	1	0	0	1	0	1
16	0	0	0	0	1	1	1	1	0	0
17	0	0	1	1	1	1	0	0	0	0
18	0	0	1	0	1	1	0	1	0	0
19	0	0	0	0	0	0	0	0	1	0

Tabella 2 Insieme delle marcature.

marcatura	transizione	marcatura
1	1	2
1	3	3
1	4	4
2	3	5
2	4	6
2	6	1
3	1	5
3	4	7
3	8	1
4	1	6
4	3	7
4	9	1
5	2	8
5	4	9
5	6	3
5	8	2
6	3	9
6	6	4
6	9	2
7	1	9
7	8	4
7	9	3
8	4	10
8	5	11
8	7	12
8	16	13
9	2	10
9	6	7
9	8	6
9	9	5
10	5	14
10	7	15
10	9	8
10	16	16
11	1	12
11	4	14
11	16	3
12	4	15
12	6	11
12	16	5
13	4	16
13	5	3
13	7	5
13	8	17
14	1	15
14	9	11
14	16	7
15	6	14
15	9	12
15	16	9
16	5	7
16	7	9
16	8	18
16	9	13
16	10	19
17	3	13
17	4	18
17	5	1
17	7	2
18	3	16
18	5	4
18	7	6
18	9	17
19	11	16
19	12	13
19	13	18
19	14	9
19	15	7

rete sicura
tutte le transizioni possono scattare almeno una volta
la rete non presenta situazioni di deadlock
rete non conservativa
rete reinizializzabile
rete limitata
rete viva
rete ciclica

Tabella 3 Grafo dei casi e proprietà della rete.

p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
1	0	0	0	1	0	0	0	1	0
1	1	0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	0	1	1
0	0	0	1	0	0	0	1	1	0

a)

Tabella 4a P-invarianti

t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14	t15	t16
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
0	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0
0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	1
1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1
0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	1
1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	1

b)

Tabella 4b T-invarianti

UN SISTEMA INTEGRATO DI CONSULTAZIONE CLINICA E GESTIONE DATI PER UN CENTRO USTIONI

Emanuela Cividati, Gabriella Pasi

Istituto di Cibernetica - Università di Milano

RIASSUNTO

In questo articolo vengono descritte le caratteristiche principali di un prototipo che è stato realizzato in ambiente UCSD PASCAL sul sistema DLT 16000 per un Centro Ustioni. In tale prototipo sono state integrate le seguenti funzionalità:

1. Gestione delle cartelle cliniche.
2. Supporto per la consultazione clinica durante la fase di terapia intensiva.
3. Supporto didattico

Dopo una breve descrizione della prima fase di analisi del problema si procede ad una presentazione dei principali aspetti implementativi.

ABSTRACT

This article describes the fundamental characteristics of a prototype to be used in a Burns Center. This prototype, realized in UCSD PASCAL, on DLT 16000, integrates the following functions:

1. Management of medical records.
2. Support of medical consultation during the period of intensive care.
3. Support of doctors training.

After a brief description of the phase of problem analysis, we present the principle aspects of the implementation.

1. INTRODUZIONE

Fra i possibili campi di applicazione dell'informatica, la medicina rappresenta un'area particolarmente adatta nell'ambito della quale introdurre l'impiego di strumenti automatici. La necessità di intervento tempestivo e preciso nel trattamento clinico delle patologie che necessitano una terapia intensiva porta a considerare la possibilità di realizzazione di sistemi di supporto alle attività terapeutiche e gestionali svolte all'interno della struttura ospedaliera, al fine di migliorarne le prestazioni ed incrementare così la possibilità di sopravvivenza dei pazienti.

Il prototipo descritto in questo articolo è stato realizzato, sulla base delle precedenti considerazioni, per un Centro Ustioni; il lavoro necessario per la sua definizione e successiva implementazione è stato svolto in collaborazione con il Centro Ustioni dell'Ospedale Cà Granda di Milano.

Questo prototipo, che costituisce il primo passo per la realizzazione di un sistema da installare nel Centro, integra le seguenti funzionalità:

1. Gestione delle cartelle cliniche
2. Supporto alle attività terapeutiche svolte in fase di terapia d'urgenza (prime 48-72 ore della degenza del paziente)
3. Supporto didattico per il "training" del personale medico e paramedico che opererà nel Centro attraverso l'utilizzo del sistema di consultazione per la simulazione di un caso reale

2. PROBLEMI DI UN CENTRO USTIONI

Un Centro Ustioni è un luogo attrezzato per accogliere il "grande ustionato"; quest'ultimo è un paziente che presenta ustioni che ricoprono più del 25% della superficie corporea, se adulto, o più del 10% della superficie corporea, se bambino.

Gravi sono gli scompensi indotti sull'organismo del "grande ustionato"; un pericolo immediato è costituito dall'instaurarsi di uno stato di shock, conseguente alla grave perdita di liquidi subita, che può portare anche alla morte del paziente.

L'intervento terapeutico sul "grande ustionato" è perciò principalmente teso a contenere l'evoluzione dello shock; esso deve quindi essere preciso e mirato ed estremamente ristretti sono i tempi entro cui decidere.

Le decisioni connesse all'attività di terapia, da una parte si servono di una notevole quantità di dati relativi al paziente (quali risultati di esami, rilevazioni cliniche, etc.), dall'altra ne producono; tutti questi dati devono essere facilmente e rapidamente accessibili.

Un approfondimento della procedura clinica ed organizzativa del Centro ha portato poi alla rilevazione che alcuni degli interventi clinici e alcuni procedimenti di valutazione si basano sull'utilizzo di formule, tabelle e regole formalizzabili.

Le precedenti considerazioni hanno portato alla valutazione della possibilità di realizzare un sistema automatico di consultazione clinica integrato con un sistema di gestione dei dati, da inserire nel Centro.

Un sistema di questo tipo può rappresentare per il medico uno strumento veloce e sicuro di verifica o di guida nelle scelte terapeutiche; inoltre, relativamente alla gestione dei dati clinici, i maggiori vantaggi derivano da una prevenzione, dalla duplicazione di informazioni e dalla possibilità di ricerche di dati rapide e mirate.
Un altro aspetto caratteristico della vita che si svolge in un Centro Ustioni è quello relativo alla formazione del personale medico e paramedico che deve essere introdotto alla procedura clinica ed organizzativa operata sul "grande ustionato".

In quest'ambito si può immaginare un'utilizzo alternativo del sistema che si è prospettato sopra. I principali vantaggi derivanti da un supporto informatico al periodo di "training" sono rappresentati dalla possibilità di autovalutazione del livello di apprendimento rispetto ad un livello base e dalla possibilità di controllo, da parte dell'educatore, del comportamento dell'allievo nei confronti di un caso clinico specifico.

3. LA FASE DI ANALISI DEL PROBLEMA

Il lavoro svolto per la realizzazione del prototipo ha avuto inizio con una fase di analisi della procedura di intervento sul "grande ustionato" adottata nel Centro; tale procedura è stata analizzata a più livelli di dettaglio sia per quanto riguarda i suoi aspetti clinici che quelli gestionali e di seguito modellata attraverso Reti di Petri.
La figura 1 mostra la rete che modella il livello più generale della procedura.

In tale rete, del tipo canali-agenzie, compaiono 5 agenzie (rettangoli), che rappresentano i principali centri di attività rilevati all'interno del Reparto, e da 5 canali (cerchi), che definiscono i flussi di comunicazione fra le agenzie.

L'agenzia PRONTOSOCCORSO raccoglie quelle attività standard che costituiscono le prime cure offerte al paziente e che permettono al medico di valutare il suo stato clinico. Da queste attività sono generati i CAMPIONI da fornire alle ANALISI e le NORME in base alle quali si comporteranno TERAPIA, ASSISTENZA ed ANALISI, finchè non diverrà necessario mutarle.

Le attività che seguono quelle di PRONTOSOCCORSO sono state raggruppate in base al tipo di esecutore e alla loro natura: in TERAPIA quelle tipiche del personale medico, in ASSISTENZA quelle del personale paramedico, in PRELIEVI quelle relative all'acquisizione dei campioni per gli esami ed in ANALISI quelle del personale dei laboratori.
ASSISTENZA ed ANALISI forniscono informazioni alla TERAPIA in base alle quali quest'ultima modifica le proprie NORME e formula RICHIESTE PER PRELIEVI ED ASSISTENZA.

Le RICHIESTE DELLA TERAPIA possono portare alla modifica delle NORME PER I PRELIEVI e delle NORME PER L'ASSISTENZA.
Il dettagliamento dei centri di attività e dei canali in figura 1 ha portato alla descrizione della procedura a livelli di maggior particolarità; attraverso l'applicazione di morfismi alla rete in figura si sono ottenuti modelli in Reti di Petri delle versioni più dettagliate della procedura.

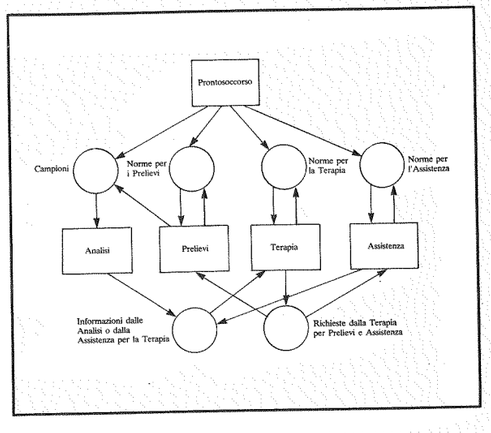


Fig. 1

Le reti ottenute per applicazioni di morfismi sono del tipo Condizioni Eventi; in figura 2 viene mostrato il morfismo operato sul Centro Prontosoccorso.

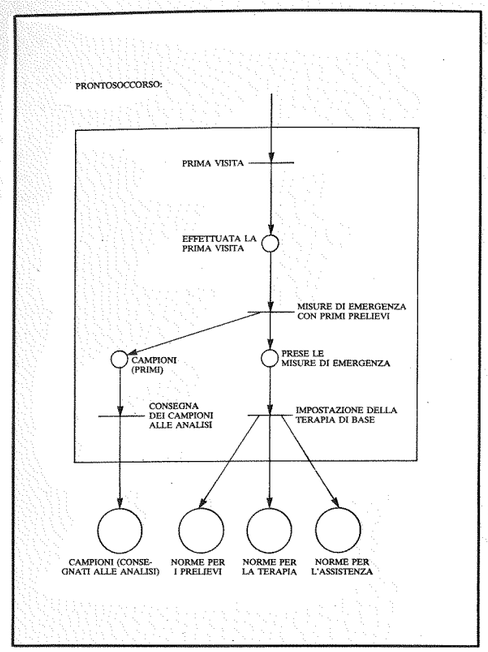


Fig. 2

In tale rete gli eventi (rettangoli) rappresentano attività o operazioni della procedura, mentre le condizioni (cerchi) sono o predicati sul flusso delle attività, o entità coinvolte nelle attività ("materiali" necessari e prodotti).
Molte delle attività identificate utilizzano e generano informazioni; le attività stesse devono per la maggior parte venir documentate.
Tutto ciò rimanda ad un flusso di informazioni che esiste sotterraneo alla rete; queste informazioni sono raccolte principalmente su documenti cartacei, moduli, che confluiscono nella cartella clinica del paziente. Abbiamo raccolto e analizzato i documenti utilizzati evidenziando le informazioni in essi contenute, momenti e luoghi di utilizzo.

Come è stato precedentemente osservato, la necessità di un pronto intervento su un paziente gravemente ustionato e l'importanza della impostazione di una corretta terapia ha portato alla decisione di realizzare una prima automazione della procedura, relativamente al periodo che copre la fase di terapia intensiva, collocata all'incirca durante le prime 48-72 ore del ricovero del paziente; gli interventi che coprono questo periodo sono tutti quelli relativi al centro Prontosoccorso e ad una prima valutazione terapeutica.

Un ultimo, importante passo nella fase di analisi ha consistito nella raccolta e strutturazione della conoscenza medica caratteristica relativa agli interventi considerati; ciò è stato realizzato attraverso lettura di materiale specialistico ed interviste al personale medico operante nel Centro dell'Ospedale Cà Granda.

4. IMPLEMENTAZIONE

Il programma è stato realizzato su elaboratore DELTA SYSTEM 16000 in ambiente UCSD PASCAL.
In questo paragrafo vengono presentate le soluzioni implementative relative ai seguenti aspetti:

a) Struttura generale del programma
b) Acquisizione e strutturazione dei dati relativi alle cartelle cliniche
c) Realizzazione del sistema di consultazione
d) Realizzazione del sistema didattico.

Struttura generale del programma

Esaurita la fase di analisi del programma e di acquisizione della conoscenza necessaria si è affrontata la fase relativa alla decisione di soluzioni implementative; occorreva realizzare un programma che fornisse un supporto agli interventi evidenziati, rispettandone le correlazioni causali e temporali e la loro collocazione nell'ambito di momenti più generali (i centri di attività nella rete in figura 1).
Inoltre, considerata la natura altamente interattiva dell'applicazione e la necessità di estrema semplicità nella comunicazione uomo-macchina, si è realizzata un'interfaccia che guida l'utente all'utilizzo del sistema in modo chiaro ed immediato; le sue principali caratteristiche sono esposte nei seguenti punti:

a) essa guida lo svolgersi delle attività in base alle relazioni causali espresse dal modello in rete;
b) data l'impossibilità di attivare esecuzioni parallele si da all'utente la possibilità, attraverso menú, di scegliere fra attività concorrenti;
c) viene cosigliata all'utente una sequenza prioritaria eventualmente modificabile relativamente all'esecuzione delle attività;

La figura 3 mostra uno dei menù proposti all'utente.

Questa schermata suggerisce all'utente che si trova nel centro di attività Prontosoccorso e può eseguire la sottoattività Prima Visita; gli interventi elencati nel riquadro in basso a destra sono quelli costitutivi della sottoattività; l'ordine in cui gli interventi compaiono stabilisce una priorità che può essere modificata dell'utente attraverso il posizionamento del cursore su un altra attività.

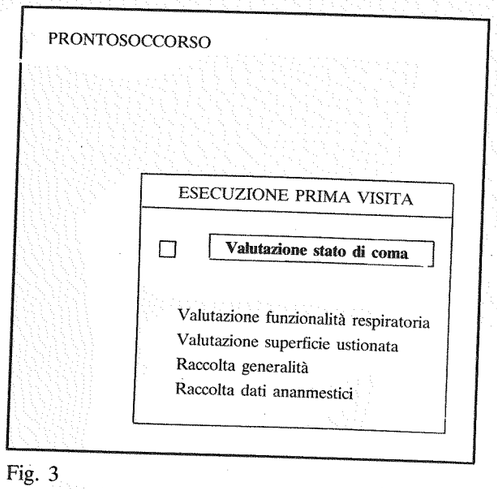


Fig. 3

L'utilizzo della funzione grafica del reverse ha un duplice significato; da un parte suggerisce l'operazione in quell'istante prioritaria, dall'altra segna il limite tra le operazioni già eseguite (le attività di Prontosoccorso vengono eseguite una sola volta) e quelle che ancora lo devono essere (le etichette delle prime stanno sopra quelle in reverse, quelle delle seconde stanno al di sotto).

Struttura e acquisizione dei dati relativi alle cartelle cliniche

I parametri principali che ci hanno guidato nelle scelte implementative relative alla strutturazione dei dati sono stati i seguenti:

- numero massimo dei pazienti contemporaneamente ricoverati nel Centro;
- tipo e frequenza dei dati da trattare.

I dati presenti nella cartella clinica sono fortemente disomogenei (valori interi, valori reali, stringhe, testi) e con frequenze di occorrenza estremamente variabili; scegliere di strutturare la cartella clinica utilizzando files sequenziali, ipotesi considerabile tenendo conto del numero ridotto dei degenti contemporanei (12 in situazioni di normalità, 37 al massimo in situazioni di emergenza) diventava, in base alle precedenti considerazioni, poco flessibile ed oneroso dal punto di vista della memoria.

Abbiamo così scelto di utilizzare TITAN, un DBMS per basi di dati relazionali realizzando nel Centro Universitario di Ginevra su Apple II in ambiente UCSD PASCAL; TITAN, il cui codice è composto da cinque unità funzionali ed un programma principale, è stato in seguito adattato al system Delta.

Analizzata la struttura di una cartella clinica adottata nel reparto si è deciso di suddividere i dati in base alla loro frequenza e ai loro momenti di rilevazione, di utilizzo e di modifica; si sono di conseguenza raccolti i gruppi di dati così ottenuti in relazioni.
La base di dati da noi creata si chiama "cartelle" e le relazioni introdotte sono 15.

Il paziente viene univocamente identificato da un numero che costituisce la chiave primaria o parte di essa di tutte le relazioni definite nella base di dati.

Il DBMS TITAN colloquia con l'utente attraverso una interfaccia propria, strutturata in menù attraverso i quali l'utente avanza al sistema le proprie richieste di creazione e utilizzo delle basi di dati; in particolare, richieste di gestione delle basi dati e delle relazioni in esso contenute vengono avanzate mediante la scelta di uno dei comandi messi a disposizione dal menù relativo, mentre il trattamento di dati organizzati nelle relazioni avviene attraverso l'utilizzo di un linguaggio non procedurale basato sul "query by example".

Poichè il nostro sistema colloquia con l'utente attraverso una interfaccia propria e, di conseguenza, la gestione della base di dati non è direttamente affidata ad esso ma deve essere mediata dal programma, si è reso necessario apportare delle modifiche al codice di TITAN, per un accesso diretto alla base di dati.

L'apertura e la chiusura della base di dati utilizzata avvengono mediante chiamata diretta da programma delle relative primitive presenti in TITAN; le richieste di inserzione, cancellazione e ricerca di tuple in una data relazione sono formulate attraverso l'utilizzo di due variabili globali, che specificano i nomi delle operazioni e i dati su cui lavorare in un formato che rispetta la struttura del DBMS. L'esecuzione della richiesta viene invocata da una procedura che contiene le chiamate alle primitive di esecuzione delle richieste di inserzione, cancellazione e ricerca di tuple. I risultati delle ricerche sono raccolti in una lista le cui componenti sono costituite dai singoli campi delle tuple.

I dati appartenenti alla cartella clinica vengono raccolti e forniti da una parte in relazione a specifiche attività terapeutiche, dall'altra attraverso la riproduzione sul video di moduli utilizzati.

Realizzazione del sistema di consultazione

Come già evidenziato precedentemente il sistema di consultazione realizzato è relativo alle attività terapeutiche intensive riconducibili all'intervallo temporale che va dal ricovero alle prime 48-72 ore successive. In particolare vengono assistite: raccolta dati anamnestici; valutazione della gravità del paziente (in base alla determinazione dello stato di coma, dell'inalazione gas e della estensione e profondità delle ustioni) con la conseguente formulazione di una prognosi; primo trattamento topico della ferita; valutazione dei dati già raccolti per gestire correttamente un trattamento di riequilibrio elettrolitico; valutazione della funzionalità cardiaca, renale e respiratoria e delle conseguenti implicazioni terapeutiche.
Le modalità di realizzazione del supporto, ovviamente diversificate in base al tipo di attività, sono riconducibili a due criteri che ora illustreremo.

Nel caso di operazioni standard generalizzabili a tutti i pazienti, quali per esempio valutazione stato di coma e valutazione superficie ustionata, non si è fatto altro che visualizzare gli schemi che vengono utilizzati durante queste attività, identificati durante la fase di acquisizione della conoscenza, compilando i quali l'utente fornisce all'elaboratore i dati necessari per un'indicazione terapeutica.

Nel caso di attività più complesse il cui svolgersi dipende fortemente da parametri caratteristici di un singolo paziente, l'interazione con l'elaboratore procede attraverso uno schema del tipo "domanda-risposta", ogni risposta permette all'elaboratore di discriminare un certo tipo di comportamento, il procedimento è sempre deterministico; esemplificative di questo caso sono tutte le valutazioni delle funzionalità.

La realizzazione di un sistema di consultazione clinica integrato con un sistema di gestione dati ha offerto la possibilità di un suo utilizzo non solo per pazienti appena ricoverati, ma anche per pazienti su cui già sono stati operati degli interventi, ma non ultimati. Ciò è stato reso possibile attraverso l'introduzione di una relazione nella quale tra le altre informazioni viene memorizzato un valore indicativo del momento in cui si è giunti nell'ambito della procedura.

Realizzazioni del sistema didattico

Come accennato in precedenza, l'implementazione di un sistema didattico è stata da noi intesa come fornire all'utente la possibilità di simulare un caso fittizio sul quale intervenire.

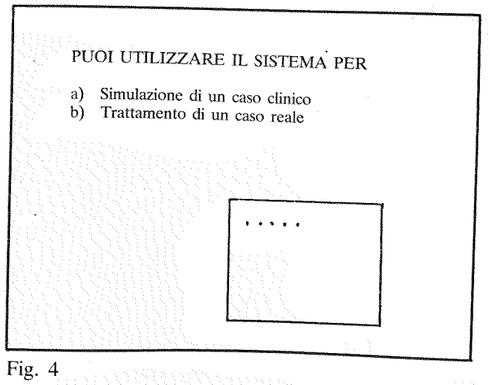


Fig. 4

Gli schemi procedurali sono i medesimi del caso reale, la differenza implementativa fondamentale è che se si opera per "simulazione" i dati non vengono memorizzati in maniera stabile nel d.b. ma viene utilizzata una cartella clinica fantoccio rimossa alla fine dell'esecuzione.
La scelta tra le due modalità di funzionamento del sistema viene effettuata dall'utente a livello della prima grata che gli viene presentata.
(vedi figura 4).

5. PROSPETTIVE FUTURE

È possibile prospettare una prosecuzione del lavoro svolto in base alle seguenti considerazioni:

a) l'approfondimento della procedura clinica svolta nel reparto e la acquisizione della conoscenza medica ad essa relativa permetterà l'estensione del sistema di consultazione oltre le assunzioni temporali fin qui fatte.

b) si prospetta possibile l'impiego di tecniche dell'intelligenza artificiale per l'automazione delle valutazioni funzionali già implementate deterministicamente e di attività terapeutiche complesse che si basano sulla acquisizione di informazioni cliniche relative al paziente; ciò consentirà l'ottenimento di prestazioni più elevate.

c) poichè l'accesso alle cartelle cliniche è sganciabile dal sistema di consultazione è possibile automatizzare la gestione dati di tutto il reparto; auspicabile è la generazione di un archivio remoto di cartelle cliniche a cui si possa accedere per rilevazioni statistiche.

d) l'utilizzo del sistema a scopo didattico può essere arricchito dalle seguenti realizzazioni:

   - dizionario della conoscenza medica al quale attingere su richiesta dell'utente
   - possibilità da parte del sistema di fornire motivazioni di una scelta compiuta.
   - utilizzo di un videodisco per la strutturazione di una banca immagini; le immagini dovrebbero venire presentate a titolo esplicativo in alcuni momenti del trattamento.

6. BIBLIOGRAFIA

[1] L. Donati: LA MALATTIA DA USTIONE, Tamburini Ed., Milano, 1975.

[2] L. Brambilla, L. Donati: LO SHOCK DA USTIONE, Ed. Minerva Medica, Torino, 1978.

[3] L. Ferrario, G. Fontanella, B. Zonta: L'IMPIEGO DELLE RETI DI PETRI NELLA DESCRIZIONE DEL COMPORTAMENTO DI UN REPARTO DI TERAPIA INTENSIVA PER GRANDI USTIONATI, Journal of Clinical Computing, 1980.

[4] L. Ferrario, B. Zonta: ANALISI DI PROCEDURE IN UN REPARTO OSPEDALIERO, Milano, Istituto di Cibernetica, aprile 1982.

[5] C.A. Petri: INTERPRETATION OF NET THEORY, GMD-ISF, 1976.

[6] J.L. Peterson: PETRI NETS, Computing Surveys, 9-3, 1977.

[7] R.S. Patil: CAUSAL REPRESENTATION OF PATIENT ILLNESS FOR ELECTROLITIC AND ACID-BASE DIAGNOSIS, Laboratory of Computer Science, Massachusetts Institute of Technology, ottobre 1981.

GESTIONE DI UN ARCHIVIO DI IMMAGINI COME AUSILIO ALLA REFERTAZIONE RADIOLOGICA

Dario Boschetti
Dipartimento di Scienze e Tecnologie Biomediche - Milano

RIASSUNTO

Questo programma serve come ausilio alla refertazione radiologica unendo, al tipico referto composto dalla lastra e da una descrizione del risultato, anche un'immagine stampata della parte del corpo interessata dall'esame con evidenziati i punti patologici. Ciò viene realizzato mediante la possibilità di creare un archivio di immagini che possono essere modificate in qualsiasi momento e mettendo a disposizione un piccolo insieme di simboli standard per evidenziare punti particolari di una figura.

ABSTRACT

This computer program acts as an aid to radiological reporting that integrates usual means such as X-ray photograph and result description with a printed image of that body parts under exam. Phatological areas are also shown. This can be made by means of an Image Data Base, allowing creation and editing at any time. Also a small set of standard graphic symbols are supported to give prominence to points of interest.

1. INTRODUZIONE

Come è noto quando viene effettuato un esame radiologico, oltre alla radiografia viene rilasciato anche un referto, cioè una descrizione di ciò che è stato riscontrato dalla lastra.
Si è pensato quindi che sarebbe utile se, unitamente alla radiografia, venisse rilasciata anche un'immagine schematizzata dell'organo interessato con evidenziate le zone danneggiate o in cui sono presenti malformazioni, (fig. 1). Per questo scopo è stato creato un programma (chiamato SimEdit) che, sfruttando un archivio composto da immagini degli organi e delle

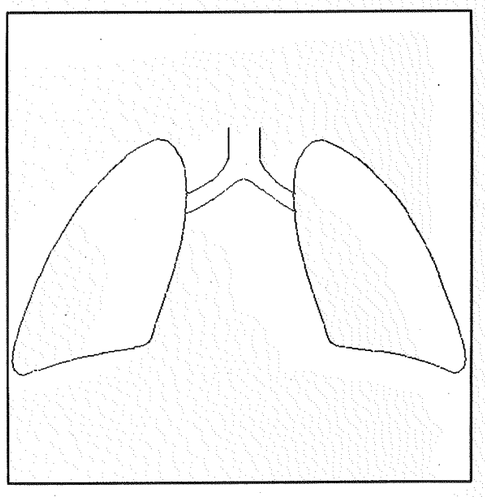


Fig. 1

ossa del corpo umano permette di stamparle; inoltre da anche la possibilità di modificare il contenuto di queste immagini o di crearne di nuove, oppure di evidenziare zone particolari con cerchi, frecce o altri simboli.

2. CARATTERISTICHE GENERALI

Questo programma è stato scritto su un DTL 16000 usando come linguaggio l'U.C.S.D. Pascal.
La caratteristica fondamentale richiesta è la semplici-

tà del suo impiego, perchè deve essere usato da medici, quindi da persone che molto spesso non hanno dimestichezza con l'uso dei computer. Per questo motivo si è deciso di utilizzare come strumento per l'interfaccia utente un mouse ottico ed il vantaggio maggiore di questo mezzo è dato dal fatto che sul video appaiono le scritte delle varie opzioni e per sceglierle basta posizionarsi col cursore mosso dal mouse sulla scritta prescelta e premere uno dei tasti, fig. 2. In particolare per facilitare al massimo l'uso del mouse stesso è sempre presente sul video un suo disegno schematizzato con le scritte di ciò che fanno i vari pulsanti.

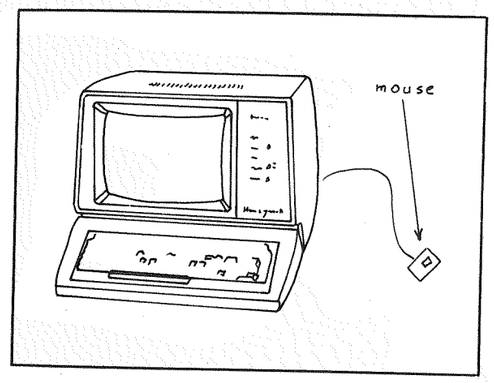


Fig. 2

Un'altra scelta è stata quella di gestire le immagini tramite composizioni di Bit Map, cioè le figure sono fatte da tante sottofigure che a loro volta sono delle matrici di boolean ognuno dei quali rappresenta un pixel cioè il più piccolo punto che può essere rappresentato sul video.

3. STRUTTURA GENERALE

3.1. Moduli esterni utilizzati

Visto che il programma si basa sulla grafica e usa come strumento interattivo il mouse, si sono dovuti utilizzare i programmi di libreria che permettono la gestione sia della grafica che del mouse stesso. Questi sono in sostanza quattro Unit: una, la "Screen Control", non contiene procedure riguardanti la grafica, ma è stata utilizzata per controlli all'interno del programma; le altre, "Mousedriv", "MyNec", "MyDump_Graphics" sono state tutte fondamentali per poter realizzare una buona interfaccia utente. Infatti Mousedriv è un driver per il mouse, cioè permette il riconoscimento degli spostamenti del cursore e quale dei tre switches è stato premuto; MyNec (versione ridotta della Unit "NEC") è un driver grafico e contiene tutte le procedure per la gestione delle Bit Maps; MyDump_Graphics contiene le procedure che permettono la stampa del contenuto della memoria grafica; le stampanti utilizzabili sono la EPSON FX_80 o la HONEYWELL SARA 11.

Come già accennato le immagini sono tutte gestite tramite Bit Maps infatti ogni figura è composta da una o più parti, ognuna delle quali è una matrice 4*4 di caratteri che a loro volta sono matrici 8*8 di bits. La scelta di usare una 4*4, cioè una misura abbastanza piccola, è stata fatta sia perchè la gestione di una unità più grande sarebbe stata lenta, sia perchè si ha l'esigenza di avere la possibilità di unire più figure fra di loro per rappresentare parti del corpo più grandi.

3.2 Archivi

Per poter gestire le immagini del corpo umano, ognuna delle quali è composta da più parti, si è associato ad ogni figura un file di record (il cui nome è quello delle figura stessa), ognuno dei quali è una porzione dell'immagine. Questi record sono formati da: un numero, che identifica la parte in figura (che verrà anche chiamata "pict_block"), le coordinate in cui farla apparire, lo zoom con cui deve essere presentata e naturalmente la Bit Map associata. Per memorizzare l'elenco delle immagini è stato creato un file di stringhe (chiamato "LIST.LIMBS") che contiene esclusivamente i nomi.

Ci è anche sembrato utile fornire all'utente un file particolare in cui memorizzare simboli di utilità generale che possono servire per completare l'immagine con indicazioni particolari, questo è stato fatto creando un altro file (chiamato SYSTEM.PICTURE) contenente i record con i nomi e le Bit Maps dei simboli.

3.3 Interfaccia utente

Per quanto riguarda l'interfaccia utente la caratteristica principale è la divisione in settori del video, ognuno dei quali ha un preciso scopo: il menù è evidenziato su una striscia larga 32 pixel divisa in 6 settori posta sulla parte bassa dello schermo (in ogni parte compare la scritta di un'opzione), in alto a destra è presente il disegno schematizzato del mouse in cui su ogni bottone è scritta la funzione che viene compiuta allo stesso e sotto questo disegno c'è una zona in cui, nei vari momenti del programma, compaiono scritte di indicazione per l'utente, il resto del video è lasciato per la gestione delle immagini e sarà solo questa parte che verrà eventualmente stampata.

Nella fase di inizializzazione viene fatto apparire il menù principale che è il seguente:

CLEAR NEW PICTURE SPEC SYMBOL PRINT EXIT-PROGRAM

e sul disegno rappresentante il mouse compare solo

la scritta "Select" sul bottone destro, cioè in questa fase è d'obbligo la selezione di un opzione, inoltre viene fatto apparire il cursore che in questo caso è una croce ("cross"); per farlo muovere, in tutte le procedure principali del programma c'è un ciclo di REPEAT in cui viene continuamente letta la posizione del mouse tramite la FUNCTION "read_tablet" contenuta in "Mousedriv" e viene aggiornata l'immagine (procedura "ob_draw") cancellando quella esistente (procedura "draw_object"), controllando che l'area occupata dal cursore non esca dallo schermo (procedura "clip_object"); viene usata "draw_object" sia per cancellare che per far apparire perchè la modalità logica di tracciamento è settata in modo che sia fatto il complemento. Quando viene premuto il bottone corrispondente a "Select", la prima cosa che il programma fa è controllare che il cursore si trovi nella zona riservata al menù, in caso affermativo riconosce l'opzione seguendo l'operazione: ascissa DIV 6 il cui risultato (variabile "comand"), che può variare tra 0 e 5, viene utilizzato in un "CASE comand OF" per eseguire l'opzione corrispondente.

4. MODALITÀ D'USO

4.1 Clear

Pulisce la zona del video riservata alle immagini chiamando la procedura "clear_nec" contenuta in "MyNec".

4.2 New

Serve per creare immagini che poi devono essere memorizzate, sia simboli che arti, infatti sulla parte dello schermo riservata alle indicazioni particolari viene posta la domanda se si vuole creare un arto ("Limb") o un simbolo ("Symbol") e successivamente se si vuole crearne uno nuovo o modificarne uno vecchio. Se si sceglie la modifica viene eseguita la procedura "which_char", questa serve per presentare l'elenco di quelle esistenti e riconoscere la scelta effettuata; "which_char", a seconda che si tratti di un "Limb" o di un "Symbol" scandisce LIST.LIMBS o il campo nome di SYSTEM.PICTURE visualizzando sei nomi alla volta; se ce ne sono altri, in parte al sesto è presentato il simbolo "›" per indicare che c'è un'altra pagina di nomi; il simbolo "›" è presentato se si è in una pagina successiva alla prima e serve per tornare alla precedente. La scelta viene riconosciuta in base al numero di pagina e all'ordinata del cursore usando anche in questo caso l'algebra modulare, si individua così il numero del record in cui si trova il nome scelto. Se si tratta di un arto viene aperto il file relativo e presentato l'elenco dei simboli che lo compongono dando la possibilità sia di farli apparire tutti (premendo il pulsante con la scritta "All"), sia solo una parte di essi selezionandoli uno a uno; è da notare che man mano che sono visualizzati, viene anche memorizzata su una lista linkata tutti i dati di posizione, zoom, nome e Bit Map di ogni simbolo, terminata la selezione ogni pict_block verrà delimitato da un rettangolo per evidenziare la zona di video occupata dallo stesso; se inizialmente era stata richiesta la creazione di una nuova immagine naturalmente apparirà solo il contorno.

4.3 Creazione e modifica di un "symbol"

In questo caso il menù è il seguente:

GRID ZOOM^{+++} ZOOM^{---} DELETE SAVE ABORT

e le funzioni del mouse sono:

SET RESET SELECT.

Le funzioni "Set" e "Reset", se il cursore si trova all'interno del rettangolo che delimita il simbolo, creano e tolgono i punti che compongono la figura, l'individuazione del punto viene fatta utilizzando l'algebra modulare cioè: prendendo come riferimento l'angolo in basso a sinistra del rettangolo, le coordinate del cursore DIV 4 mi danno il carattere interessato alla modifica, mentre le coordinate MOD 8 mi danno il punto interno al carattere; l'aggiornamento della Bit Map viene fatta con la procedura set_reset.

- GRID: fa apparire un reticolo che evidenzia ogni unità fondamentale che compone la figura.
- ZOOM^{+++}, ZOOM^{---}: servono per zoommare il simbolo.
- DELETE: serve per cancellare il simbolo e toglierlo da SYSTEM.PICTURE, in questo caso viene vuotato il record relativo e viene fatto un Kranch dell'arry che forma il file.
- SAVE: salva la nuova Bit Map e richiede (se si tratta di un modifica) se il nome debba essere cambiato (se il Symbol è nuovo richiede direttamente l'introduzione del nome) in caso affermativo il simbolo vecchio non viene eliminato, ma ne viene creato un altro col nuovo nome e le modifiche apportate a quello vecchio.
- ABORT: annulla tutte le modifiche e torna al menù principale.

4.4 Creazione e modifica di un "symbol"

In questo caso il menù è il seguente:

GRID OLD PICTURE NEW BLOCK DELETE SAVE ABORT

le funzioni del mouse sono:

SET RESET TAKE.

e compare anche la scritta: "Any button for select picture" per indicare che per selezionare un'opzione si può premere qualsiasi tasto; se si crea un nuovo arto inizialmente appare solo il contorno di un pict_

al centro del video. Premendo "Take" se si è sopra un pict_block questo diventa il cursore e si ha così la possibilità di spostarlo; le funzioni diventano:

POSITION ZOOM^{+++} ZOOM^{---}

e servono per zoommare e posizionare il simbolo, quando si posiziona tornano ad essere presenti le funzioni precedenti.

- GRID: compare la richiesta di selezionare il pict_block su cui si vuole far apparire il reticolo, a questo punto basta selezionare il simbolo.
- DELETE: compare la richiesta di selezionare quale pict_block si vuole cancellare, soddisfatta la richiesta il simbolo scompare dal video e viene anche tolto dalla lista presente in memoria.
- SAVE: anche in questo caso viene chiesto se si vuole cambiare nome (in caso di modifica di un arto preesistente, se l'arto è nuovo si richiede direttamente il nome) e successivamente appare questa richiesta anche per ogni pict_block componente l'arto, evidenziando la parte interessata complementandola; terminata l'introduzione dei nomi la memorizzazione viene effettuata scandendo la lista e trasferendo tutti i dati sul nuovo file col nome dell'arto, anche in questo caso si trattava di una modifica e si è dovuto cambiare nome, quello vecchio non viene cancellato, ma ne viene creato uno nuovo lasciando inalterato il precedente.
- ABORT: torna al menù principale senza prendere in considerazione le modifiche effettuate.

4.5 Picture

Questa opzione serve per far apparire immagini, come per la "new" viene richiesto se si vuole "Limb" o "Symbol" e poi appare l'elenco, anche in questo caso viene utilizzata la lista per memorizzare tutti i dati (se si tratta di arti viene memorizzato anche il nome dell'arto a cui appartengono) dei pict_block.

4.6 Spec simbol

Il menù diventa:

ERARER ARROW CROSS CIRCLE RETTANG EXIT

e le funzioni:

LINE TAKE SELECT

– LINE: premendo questo tasto le funzioni diventano:

PENCIL RUBBER EXIT

con "Pencil" il cursore diventa una matita e viene lasciata una traccia di ogni movimento del mouse, nel frattempo sul bottone centrale è apparsa la scritta "Stop" che serve per fermare la scrittura; con la funzione "Rubber", prendendo come inizio il punto in cui è stato premuto il bottone, compare un segmento che congiunge il punto di riferimento alla posizione attuale del cursore, anche in questo caso ci si ferma premendo il tasto su cui è apparsa la scritta "Stop"; "Exit" serve per tornare alle funzioni precedenti.

- TAKE: se si è sopra un pict_block questo diventa il cursore e si ha così la possibilità di spostarlo. Nel caso faccia parte di un arto si controlla se il file associato è già aperto, nel caso il file attualmente in memoria sia diverso viene chiuso ed aperto quello giusto; questo per minimizzare gli accessi a disco.

  Le funzioni diventano:

  POSITION ZOOM

  con gli evidenti significati.

La selezione delle varie opzioni serve essenzialmente per cambiare il cursore, infatti:

- ERASER: il cursore diventa una gomma e le funzioni sono:

  DELETE SELECT

  premendo "Delete" viene cancellato tutto quello su cui passa il cursore, la cancellazione viene interrotta premendo il solito tasto di "Stop".

- ARROW: ora il cursore è una freccia e le funzioni:

  POSITION TURN ANGLE SELECT

  con "Position" rimane impresso il cursore sul video, mentre con "Turn Angle" viene cambiato il verso della freccia di 45 gradi in senso antiorario.

- CROSS: il cursore torna ad essere una "cross" che è l'unico con cui si possono spostare pict_blocks.

- CIRCLE: il cursore diventa un cerchio e le funzioni:

  POSITION RUBBER SELECT

  "Position" fa rimanere impressa l'immagine del cerchio, mentre con "Rubber" si può variare il suo raggio che verrà fissato col solito "Stop".

- RETTANG: il cursore diventa un rettangolo e le funzioni:

  POSITION RUBBER SELECT

  in questo caso con "Rubber" si modificano le dimensioni del rettangolo.

- EXIT: il programma torna al menù principale e il cursore diventa automaticamente la "cross".

È da notare che quando il cursore è un pict_block o la gomma in fase di cancellazione, si può muovere solo all'interno della zona predisposta per la gestione delle immagini, questo perchè nel primo caso l'eventuale selezione di un'opzione provocherebbe la scomparsa della figura, nel secondo perchè cancellerebbe parti della maschera o scritte del menù.

4.7 Print

Serve per stampare le immagini; viene richiesto se la stampante collegata è una EPSON o una SARA 11, soddisfatta la richiesta la Bit Map presente nel NEC viene prima trasferita su un file di supporto chiamato "TEMP" e poi da questo alla stampante.

4.8 Exit Program

Con questa opzione si esce dal programma, vengono chiusi i files LIST.LIMBS e SYSTEM.PICTURE e viene inoltre ripulito il video grafico.

5. NOTE CONCLUSIVE

Questo programma è ancora a livello di prototipo, comunque è stato scritto in modo che sia facilmente modificabile e l'aggiunta di una nuova opzione o la modifica di una già esistente è molto semplice da effettuare. È molto probabile infatti che debbano essere apportati cambiamenti, dovuti soprattutto a suggerimenti di chi lo dovrà utilizzare.
Si è voluto sostanzialmente creare uno strumento che dia la base per lo sviluppo di un nuovo metodo di refertazione più semplice e chiaro tenendo anche conto che può essere collegato con un programma di refertazione automatica.

6. BIBLIOGRAFIA

[1] PASCAL SYSTEM DELTA D.1 PROGRAM'S MANUAL, Delta Instrumentation, ottobre 1983

[2] M. Wirth, ALGORITHMS + DATA STRUCTURES = PROGRAMS, Prentice Hall, Inc., 1976

[3] J.D. Foley, A. Van Dam, FUNDAMENTALS OF INTERACTIVE COMPUTER GRAPHICS, Addison Wesley Publishing, 1983