Honeywell
Honeywell Information Systems Italia
Via Laboratori Olivetti - Pregnana Milanese

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA
Via Viotti, 5 - Milano

4

NOTE DI SOFTWARE

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA

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 dell'Istituto di Cibernetica 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 o 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.
La preparazione del testo nel formato adatto alla fotoriproduzione è a carico degli autori, ai quali verranno comunicate le norme redazionali.
I contributi non dovrebbero superare le quattromila parole.

I lavori accettati per la pubblicazione non vengono revisionati da un comitato tecnico. Pertanto, ogni contributo pubblicato riflette le opinioni personali 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 a chi ne faccia richiesta alla redazione.

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

4

FPDM-110

Ottobre - Dicembre 1977

NOTE DI SOFTWARE

Indice:

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

QUESTO NUMERO . . .

Proseguendo nel dibattito, avviato nello scorso numero, sui problemi e sulle tecniche della documentazione del software, NOTE DI SOFTWARE 4 propone tre nuovi contributi legati a questo tema.

Il primo (Degli Antoni e Zonta) analizza l'uso di procedimenti metaforici nell'attività di documentazione del software e, più in generale, nella scienza degli elaboratori.

Il secondo (Cappelli, De Antonellis, Mazzocchi e Polillo) presenta un linguaggio grafico ("schemi di dialogo"), analogo per molti aspetti al linguaggio dei flow-charts. Tale linguaggio è particolarmente adatto alla descrizione dei dialoghi che si realizzano, in un'applicazione transazionale, fra sistema centrale e operatore al terminale.

Il terzo (Cavallari, Pedrotti, Tisato) propone l'uso del SIMULA 67 come linguaggio di documentazione e di specifica. In tal modo, il sistema in via di sviluppo può venire eseguito, e verificato nella sua struttura logica, prima ancora che ne inizi la implementazione definitiva.

PHOS (Program Hierarchy from Output Structure), una metodologia di programmazione particolarmente adatta allo sviluppo di applicazioni EDP, è il tema di altri due lavori coordinati. Il primo (Bettinazzi, Maiocchi, Maserati, Monzani, Spoletini e Toscani) illustra le linee fondamentali dei vari passi della metodologia, seguendo i quali il programmatore può "dedurre" la struttura del programma direttamente dalla struttura dei risultati ad esso richiesti (tenendo conto, in modo opportuno, anche della struttura dei suoi dati di ingresso).

Il secondo (Bettinazzi, Maserati, Monzani e Toscani) presenta un esempio tipico di applicazione di PHOS: lo sviluppo, passo passo, di un programma che fornisce il rendiconto dei movimenti relativi a un insieme di articoli, dall'analisi iniziale via via fino alla codifica completa in Cobol.

La Redazione

DISCUSSIONI

METAFORA E DOCUMENTAZIONE

G.Degli Antoni(°) - B.Zonta(+)

1. Metafore e comprensione del testo.

Documentare è un verbo che indica una complessa attività rivolta a produrre documenti, ovverosìa a costruire testi che diano al lettore la possibilità di comprendere la struttura, il modo di funzionare, le funzioni, le applicazioni, ..., di un qualche sistema (ospedale, elaboratore elettronico, procedura organizzativa, eccetera).

Generalmente, la documentazione è prodotta da umani ed è diretta ad esseri umani, e ciò anche nel caso che alla documentazione sia associata una qualche attività meccanizzata.

La documentazione richiede una vasta attività concettuale, sia da parte di chi documenta sia da parte di chi legge o utilizza il documento.

Questa attività concettuale fa ricorso o può far ricorso a tutti i mezzi intellettuali di chi documenta e di chi legge. E fra questi certo vi è la capacità di estrarre informazioni, di confrontarle, di costruire analogie, eccetera.

Tutti questi mezzi vengono impiegati sia nella fase della trasmissione sia nella fase della ricezione dei messaggi: ad esempio si potrà documentare impiegando direttamente analogie o facilitando l'evocazione di analogie; e si potrà comprendere il testo attraverso analogie o confronti, anche se questi non sono stati impiegati esplicitamente.

La capacità di evocazione del testo risulterà tanto più alta quanti più saranno i concetti associati alla lettura del testo stesso.

(°) - Istituto di Cibernetica dell'Università di Milano
(+) - CNR, Centro di Cibernetica e Attività Linguistiche, Milano

La possibilità di evocare immagini o altri concetti rende più semplice la comprensione e il ricordo del testo anche nel caso di documentazione tecnica, la quale è spesso rivolta alla definizione formale di sistemi.

La capacità evocativa della documentazione formale la rende inoltre accettabile anche a chi non è abituato a trattare concetti secondo rigorose definizioni formali.

Ma nella documentazione di sistemi complessi ed in particolare in sistemi (*) di elaborazione delle informazioni, la documentazione trasmette concetti che almeno in linea di principio sono ignoti al destinatario.

Questa situazione è particolarmente chiara anche nel caso del software e più specificatamente nel caso di programmi.

Il lettore sarà cioè in grado di comprendere i costrutti della documentazione, sarà in grado di associare una sua interpretazione a parti, ma potrebbe non essere in grado di comprendere il significato complessivo del testo a partire dall'analisi di sue parti. Ed infatti molto spesso i concetti trasmessi saranno nuovi per il lettore (non ci stiamo riferendo ad una originalità astratta, misurata con il metro accademico), talvolta anche semplici, ma spesso nati in un ambito applicativo con nuovi obiettivi. In questi casi il compito della documentazione è complesso: introdurre ed illustrare nuovi concetti.

Non è certo tuttavia l'informatica la sola disciplina a porsi problemi di quel genere. Si può dire che tutte le discipline costruttive e tutte le tecnologie hanno dovuto affrontare il compito di illustrare le innovazioni in modo convincente, o semplicemente hanno dovuto illustrare le stesse innovazioni, pena la loro non accettazione.

E dunque a problemi di innovazione sono spesso (forse anche sempre) associati problemi di comunicazione.

Può sembrare facile comunicare nuovi concetti. Ma non ci vuole molto a convincersi che il problema è tutt'altro che banale. Ed in effetti molto poco è stato detto su questo tema dalle varie discipline che si occupano della comunicazione umana. Ma dove è la difficoltà? Per illustrarla consideriamo due esempi.

. Dapprima l'esempio della introduzione di tecnologie che permettono l'utilizzazione di un sistema di calcolo da parte di più utenti, in modo che ciascuno di essi abbia sostanzialmente l'impressione di essere il solo utente.

(*) Il termine sistema qui come altrove in questo testo è impiegato nella accezione generale.

Come è noto, esistono vari meccanismi con cui ciò è possibile. Questi si differenziano soprattutto per il modo con cui il tempo di elaborazione viene assegnato ai vari utenti. Come descrivere brevemente il risultato, ovvero come descrivere questa forma di interazione con un elaboratore da parte di più utenti?

. L'introduzione di meccanismi di controllo del traffico stradale ha reso possibile l'adozione di sistemi capaci di sincronizzare i semafori al fine di ottenere un flusso di veicoli ordinato.
Anche in questo caso, come identificare brevemente il risultato?

In entrambi i casi, è stata scelta l'adozione di termini dal linguaggio ordinario capaci di evocare metaforicamente il risultato. Nel primo caso è stato impiegato il concetto di "condivisione di tempo" per evocare gli aspetti principali della interazione. Nel secondo caso si parla spesso di "onda verde".

La linguistica ha da sempre affrontato lo studio delle metafore come una delle basi della dinamica evolutiva delle lingue.

L'analisi di alcuni concetti della linguistica nel caso di costrutti metaforici introdotti dalla tecnologia ed in particolare dalla scienza degli elaboratori è lo scopo di questa nota.

2. Metafore e scienza degli elaboratori.

La scienza degli elaboratori propone molto spesso innovazioni. E spesso, quindi, tali innovazioni propongono l'adozione di termini metaforici. Poichè, inoltre, le applicazioni degli elaboratori si vanno estendendo sempre più, è anche sempre più frequente l'introduzione di spiegazioni che fanno direttamente ricorso a termini antropomorfi o comunque legati alla esperienza comune. E l'adozione di tali termini è legata ad ovvie estensioni del significato degli stessi o ad un loro significato metaforico.

La possibilità di distinguere fra estensione e metafora nell'uso di un termine non è sempre evidente, e dipende dalla familiarità dei concetti involti e dall'abitudine evocativa del soggetto, in un assegnato contesto. Così ad esempio il termine "coda", di impiego molto frequente nella scienza degli elaboratori, può essere visto in due modi:

a) Come un modo ovvio di indicare un certo ordinamento, in questo caso quello dei dati in una serie di registri.
b) Come il particolare nome assegnato a un certo ordinamento di dati in una serie di registri (nome trasferito da esperienze note: code di attesa alla fermata di un autobus, ec-

cetera).

La stessa situazione può essere avvertita nel caso del termine "semaforo", anche se verosimilmente in questo caso può sembrare più convincente l'interpretazione metaforica (o da trasferimento).

E' legittimo chiedersi se la scienza e la tecnologia degli elaboratori differiscono da altri campi di attività umana rispetto all'introduzione di termini che possono essere interpretati come metaforici (termini metaforici).

La diffusione dei termini metaforici dipende moltissimo dal loro impiego. Nel caso di termini metaforici imposti dal costruttore ad un particolare sistema di calcolo, la diffusione del termine sarà legata alla diffusione del sistema stesso nell'ambito degli utenti, e dipenderà sia dal"successo" della funzionalità implicita nell'uso del termine che dalla "felicità" (così dicono i linguisti) dell'espressione nell'ambito di tecnici che adotteranno tali termini per altre realizzazioni.

Da questo punto di vista, dunque, il meccanismo di diffusione dei termini metaforici nella scienza degli elaboratori, sembra essere dello stesso tipo di quello dei termini metaforici associati a qualsiasi altra tecnologia.

Forse la differenza principale rispetto ad altri campi sta, come già segnalato, proprio nella natura antropomorfa delle applicazioni degli elaboratori e dei meccanismi che le rendono possibili.

Nelle applicazioni di intelligenza artificiale, ad esempio, l'impiego metaforico di termini è così esteso da aver favorito la nascita di una disciplina (proprio la Intelligenza Artificiale) che fra l'altro si propone di rendere possibile la realizzazione di sistemi intelligenti.

Ma anche nelle applicazioni non direttamente antropomorfe, ad esempio in quelle relative ai sistemi operativi, l'impiego di termini metaforici è molto vasto, anche se talvolta la soluzione di particolari problemi propone l'adozione di termini con significato metaforico limitato (ad esempio "spooling").

Per i linguisti la fonte dell'innovazione va cercata in centri di prestigio. Al di là del significato accurato del termine prestigio, anche la scienza degli elaboratori ha i suoi. A tali centri di prestigio è spesso associata l'introduzione e la diffusione di termini metaforici. Ciò non significa che l'origine degli stessi sia da attribuirsi a ipotetiche autorità; infatti, come osserva un esperto del '700, "si fanno più metafore in un giorno di mercato che in molti giorni di assemblee accademiche".

3. Impiego di termini metaforici nel software.

E' stato più volte osservato che l'introduzione delle metafore ha conseguenze creative. La ragione di tale affermazione può essere fornita brevemente se si accetta l'ipotesi che l'introduzione di un termine metaforico implica una analogia fra sistemi differenti o fra campi differenti. In questo caso, infatti, l'ingresso di un termine metaforico suggerisce l'analisi di altri termini dello stesso sistema o campo.

Così, ad esempio, l'adozione di un termine come "semaforo" potrebbe suggerire anche la adozione di "pulsante per i pedoni". Tale pulsante per i pedoni potrebbe venir impiegato per rappresentare che certi utenti di un sistema di calcolo (programmi) hanno la possibilità di cambiare a loro piacimento la direzione del traffico.

Ancora: se si pensa che i semafori siano inseriti in una rete di semafori sincronizzati da meccanismi che permettano la realizzazione di "onde verdi", si può ragionevolmente supporre che una rete di elaboratori, ciascuno capace di assolvere a determinate funzioni, sia connessa attraverso semafori. I meccanismi di onda verde altro non rappresenterebbero che quei meccanismi per massimizzare il traffico (di programmi) attraverso le varie risorse (che nell'analogia stradale sono rappresentate dai tratti stradali fra i semafori).

La metafora introdotta non è che una delle tante possibili per analizzare le funzionalità di sistemi di calcolo. Ma altre possono essere promosse per analizzare il modo con cui i programmi sono costruiti, indipendentemente dalle loro funzionalità.

Ad esempio, è ben noto il concetto di "information hiding". Secondo tale concetto, il modo di operare di parti del programma è nascosto in modo opportuno. Se si considera il termine "hiding", in senso metaforico, verosimilmente ci si possono porre molte domande capaci forse di indicare nuovi atteggiamenti nella costruzione di programmi. Ad esempio, se pensiamo che il meccanismo di "hiding" sia una sorta di abito che nasconde (tutto o una) parte del programma, allora possiamo anche immaginare di poter nascondere (vestire) parti differenti con altri meccanismi che potrebbero dipendere dall'uso o dal momento dell'uso, ecc.

Nel caso di programmi, l'introduzione di metafore non è particolarmente difficile, anche se è difficile introdurre termini metaforici che individuino un campo i cui termini abbiano tutti un utile significato nell'ambito della problematica in esame.

Tuttavia, almeno in fase di analisi del problema, la introduzione di termini metaforici ha un accurato significato esplorativo, anche se alla fine nulla o poco dovesse rimanere di tale analisi.

CONTRIBUTI TECNICI

SCHEMI DI DIALOGO: UNO STRUMENTO PER LA RAPPRESENTAZIONE DEI DIALOGHI IN UN SISTEMA INTERATTIVO

S.Cappelli(°), V.De Antonellis(°), S.Mazzocchi(°+), R.Polillo(°+)

1. INTRODUZIONE

Il presente lavoro si colloca nell'ambito dei problemi di documentazione (e di disegno) di programmi applicativi di tipo transazionale.

In [1], Denert propone una semplice notazione grafica, basata su diagrammi di stato, per descrivere i dialoghi che si realizzano in un "dialogue system". Tali diagrammi sono adatti a descrivere un'ampia classe di dialoghi: dialoghi che si realizzano fra operatore al terminale e elaboratore centrale, dialoghi fra più elaboratori interconnessi, ecc.

Un particolare sistema per la realizzazione di applicazioni transazionali è TPS (Transactional Program System), sviluppato nei laboratori HISI di Pregnana Milanese per il Livello 62. [2]

In esso, ogni transazione è realizzata da un opportuno insieme di "message routines", scritte dall'utente. Ogni messaggio in input da un terminale viene ricevuto e identificato da un monitor sempre residente il quale provvede a (richiamare in memoria e) innescare la message routine adibita al suo trattamento. Questa, dopo l'elaborazione del messaggio, fornisce al terminale coinvolto nella transazione un opportuno messaggio di output che potrà essere o un messaggio"informativo", oppure una "domanda" alla quale il terminalista dovrà rispondere con un successivo messaggio di input, che innescherà a sua volta la message routine opportuna.

In questa nota, gli schemi di Denert, qui chiamati "schemi di dialogo", vengono specializzati a una classe particolare di dialoghi, suggerita dalle scelte effettuate dai progettisti di TPS. Tale classe coinvolge due interlocutori con ruoli differenti, e predefiniti: un interlocutore ("sistema") pone domande, mentre l'altro("terminalista") risponde al primo, ed è motivata dal fatto che, come è mostrato più oltre mediante un semplice esempio (prenotazione posti aerei), ben si presta a modellare i dialoghi che si realizzano nell'ambito di applicazioni transazionali.

Dopo aver introdotto, informalmente, la classe di dialoghi in esame e illustrato con un semplice esempio la nozione di "schema di dialogo" (par.2), si presenta la notazione proposta per la specifica dei vari tipi di messaggi (par.3), si danno le regole di composizione degli schemi di dialogo (par.4), e si indica la possibilità di corredarli di asserzioni (par.5). Si sviluppa poi un semplice esempio (par.6) e si individuano, in esso, delle strutture di dialogo "tipiche" (par.7). Alcune conclusioni sull'utilità della notazione vengono tratte nel par. 8.

2. DIALOGHI E SCHEMI DI DIALOGO

Prima di definire rigorosamente la notazione proposta, specifichiamo informalmente la classe di dialoghi a cui siamo interessati. Tali dialoghi hanno le seguenti caratteristiche:

. ogni dialogo avviene fra due interlocutori, che verranno detti "interlocutore-sistema"

(+) H.I.S.I. - Pregnana Milanese

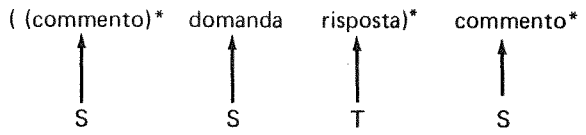
(°) Ist. di Cibernetica, Università di Milano

(S) e "interlocutore-terminalista"(T);

. S e T dialogano scambiandosi messaggi. Questi sono classificati in: domande, risposte e commenti (il termine 'commento' si riferisce, per ora, semplicemente a quei messaggi che non appartengono a nessuna delle altre due classi);

. i ruoli dei due interlocutori S e T sono prefissati, e differenti: S può porre solo domande, e inviare commenti, mentre T può fornire solo risposte;

. la struttura del dialogo fra S e T è alternata, come segue(°):



Cioè, ogni dialogo inizia con una domanda (opzionalmente preceduta da commenti) e termina con una risposta (opzionalmente seguita da commenti), e si svolge come una successione alterna di domande/risposte alla precedente domanda.

Ad esempio, un dialogo della classe considerata è il seguente:

BUONGIORNO	CHE COSA DESIDERA?	NOCI
commento	domanda	risposta
S	S	T

QUANTI ETTI? 3		ECCO LE NOCI
domanda	risposta	commento
S	T	S

CHE COSA DESIDERA?	NULLA	ALLORA ARRIVEDERCI
domanda	risposta	commento
S	T	S

• Abbiamo così introdotto, sia pure informalmente, il concetto di dialogo, inteso come la sequenza attuale dei messaggi scambiati, nel tempo, fra due interlocutori. Questo concetto, tuttavia, non è sufficiente: se noi vogliamo descrivere in modo completo le interazioni fra due interlocutori (quindi, in particolare, fra S e T) dovremo fornire non tanto (o non solo) l'effettivo dialogo che si sviluppa durante un particolare "incontro", quanto piuttosto un insieme di regole che ci permettano di ricavare tutti i dialoghi effettivamente possibili; in altre parole, dovremo fornire uno schema di dialogo. (Allo stesso modo, per descrivere un processo di calcolo non è sufficiente fornire una singola sequenza di esecuzione: ciò che occorre è piuttosto un programma che ci descriva, in modo sintetico, tutte le possibili sequenze di esecuzione effettivamente realizzabili).

Le regole rigorose per la costruzione di uno schema di dialogo verranno date nel prossimo paragrafo. Esaminiamo, per ora, un semplice esempio (fig.1)(°).

La figura si legge in questo modo: l'interlocutore S inizia il dialogo inviando un commento ('BUONGIORNO') e la domanda iniziale ('CHE COSA DESIDERA?'). Dopodichè, il turno passa a T (questo fatto è schematizzato dal simbolo ◯), il quale può rispondere o 'NULLA', o 'MELONI', o 'BANANE', o 'NOCI', oppure fornire un'altra risposta (tutte queste risposte possibili sono mutuamente esclusive). Dopo che una di queste risposte è stata fornita, il turno è di nuovo a S (questo fatto è simbolizzato da □) e così via.
Si noti che lo schema di dialogo indicato prevede dei cicli: ad esempio, se il numero di meloni richiesti da T supera la disponibilità, T viene avvisato, dopodichè la domanda 'QUANTI?' viene riproposta.

Come si vede dall'esempio, uno schema di dialogo è un grafo orientato, costituito da nodi di due tipi (◯ e □), collegati in modo alterno. Ogni diverso ◯ (o □) rappresenta un diverso "stato di dialogo" dell'interlocutore T (o S, rispettivamente).

Ogni arco del grafo è etichettato con un descrittore di una classe di messaggi (com-

(°) La notazione $(x)^*$ indica la ripetizione del simbolo x un numero n arbitrario di volte ($n \geq 0$).

(°) In figura 1, per maggiore chiarezza, sono stati omessi alcuni dettagli sintattici che verranno introdotti nel paragrafo successivo.

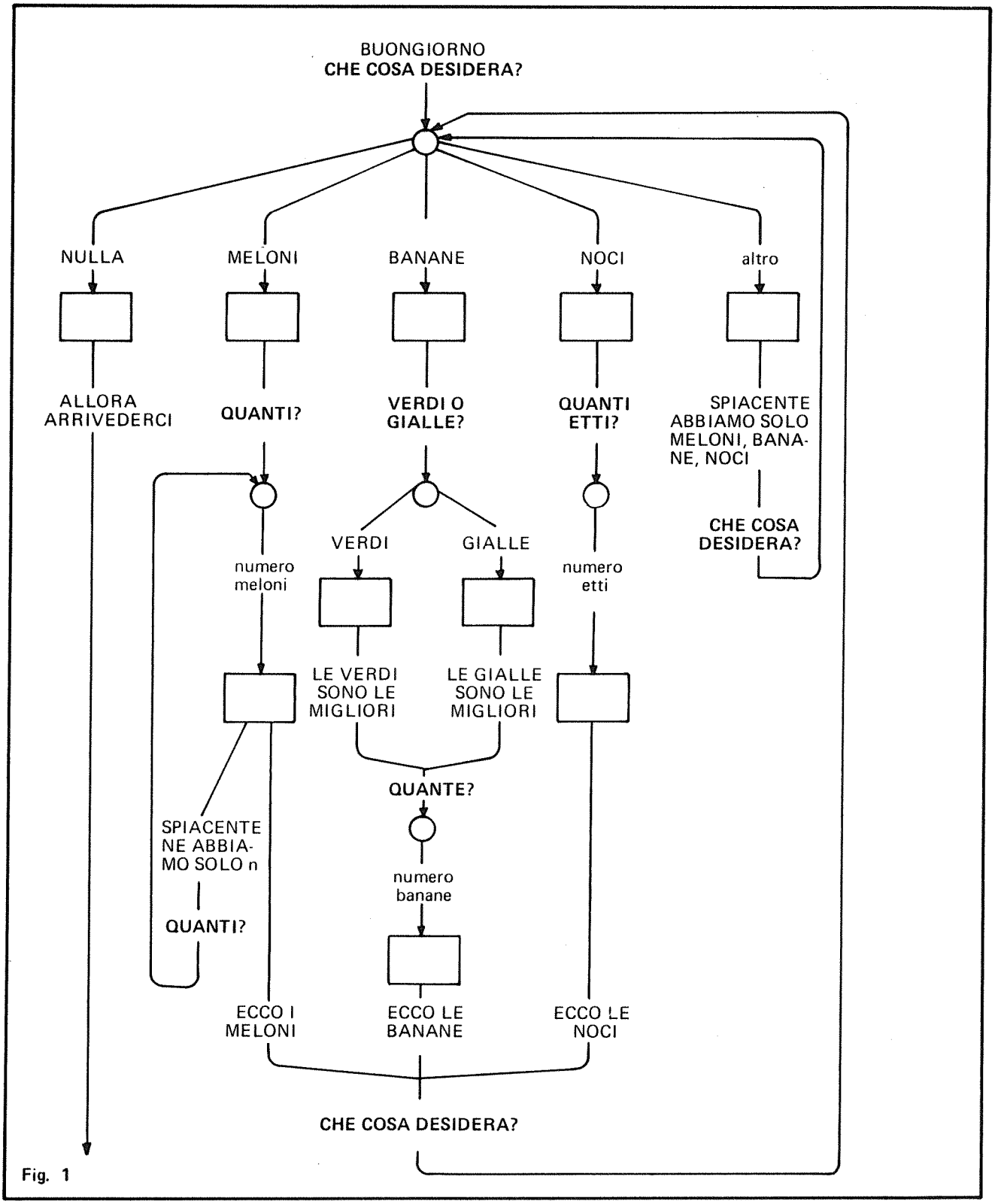

BUONGIORNO
CHE COSA DESIDERA?
NULLA
MELONI
BANANE
NOCI
altro
ALLORA ARRIVEDERCI
QUANTI?
VERDI O GIALLE?
QUANTI ETTI?
SPIACENTE ABBIAMO SOLO MELONI, BANANE, NOCI
CHE COSA DESIDERA?
VERDI
GIALLE
numero meloni
numero etti
LE VERDI SONO LE MIGLIORI
LE GIALLE SONO LE MIGLIORI
QUANTE?
SPIACENTE NE ABBIAMO SOLO n
QUANTI?
numero banane
ECCO I MELONI
ECCO LE BANANE
ECCO LE NOCI
CHE COSA DESIDERA?

Fig. 1

menti, domande, o risposte). Ad esempio, il descrittore ('SPIACENTE, NE ABBIAMO SOLO n') descrive la classe di tutti i commenti del tipo 'SPIACENTE, NE ABBIAMO SOLO 1', 'SPIACENTE, NE ABBIAMO SOLO 2', eccetera, mentre il descrittore 'SPIACENTE, ABBIAMO SOLO MELONI, BANANE, NOCI' descrive una classe costituita da un solo commento.

Nei prossimi due paragrafi, descriveremo con maggior precisione la notazione proposta, introducendo:
- la notazione per specificare i descrittori di commenti, domande, risposte e,
- le regole sintattiche per la costruzione degli schemi di dialogo.

3. DESCRITTORI DI MESSAGGI

Descriviamo separatamente la notazione per i descrittori di commenti, di domande e di risposte, utilizzando semplici esempi. Tale notazione potrà essere arricchita e standardizzata, caso per caso, a seconda dei particolari ambiti applicativi.

a) - Descrittori di commenti

Esempi di descrittori di commenti sono i seguenti:

1. 'BUONGIORNO'
2. 'LA RISPOSTA', risposta, 'È SCORRETTA. RIBATTERE'
3. 'I DATI ANAGRAFICI DEL CLIENTE SONO:', nome, cognome, indirizzo
4. 'I DATI ANAGRAFICI DEL ',qualifica,' SONO':, nome,cognome,indirizzo
5. 'SALDO=', credito(record(chiave)) - debito(record(chiave))

Come si vede, un descrittore di commento è costituito da una serie di espressioni, separate da virgole, costituite da:

- messaggi costanti (stringhe di caratteri maiuscoli racchiuse fra apici)
- nomi di variabili (stringhe di caratteri minuscoli)
- nomi di funzioni (stringhe di caratteri minuscoli sottolineati, ad esempio credito (...), record (...), oppure operatori espressi nella solita forma, ad esempio "-").

Il commento effettivamente inviato all'interlocutore T sarà allora la stringa di caratteri che si ottiene sostituendo, nel descrittore, ad ogni nome di variabile il valore da essa posseduto all'istante dell'invio, e valutando le eventuali funzioni indicate nel descrittore stesso. Ad esempio, in 5., se supponiamo che:

- record(...) sia la funzione che preleva il record la cui chiave è indicata come argomento;
- credito(...) e debito(...) siano due funzioni che selezionano, rispettivamente, il campo "credito" e "debito" del record passato come argomento,

e se supponiamo che, a un certo istante, il valore di "chiave" sia 'ROSSI' e che il campo "credito" e "debito" del record 'ROSSI' siano rispettivamente 250.000 e 150.000, il contenuto inviato sarà:
5'. 'SALDO=100.000'.

Si noti che, mediante l'uso di variabili di servizio in un descrittore, è possibile comporre commenti in modo molto flessibile: ad esempio, in 4., la variabile "qualifica" potrà essere posizionata dall'interlocutore-sistema, di volta in volta, al valore 'CREDITORE', 'DEBITORE', 'CLIENTE', ecc., per comporre, rispettivamente:

'I DATI ANAGRAFICI DEL CREDITORE ecc.'
'I DATI ANAGRAFICI DEL DEBITORE ecc.'
'I DATI ANAGRAFICI DEL CLIENTE ecc.'

b) - Descrittori di domande

Esempi di descrittori di domande sono:

1. 'NOME=', <nome>
2. 'DARE NOME E INDIRIZZO DEL CLIENTE', <nome>,<cognome>,<via>, <numero>,<città>
3. 'NOME=',<nome>, 'COGNOME=',<cognome>
4. 'QUALE L'INDIRIZZO DEL CLIENTE', nome-cliente, '?',<via>,<numero>,<città>
5. 'IL CLIENTE', nome(record(chiave)), 'HA SALDATO IL CONTO?',<risposta>

Come si vede, anche in un descrittore di domande possono comparire messaggi costanti, nomi di variabili e nomi di funzioni, individuati mediante le stesse convenzioni tipografiche usate per i descrittori di commenti. A questi elementi sintattici, si aggiungono i nomi delle variabili alle quali verrà attribuito il valore fornito in risposta alla domanda (racchiusi fra parentesi acute, ad esempio<nome>,<risposta>,ecc.). Almeno uno di questi elementi dovrà comparire in ogni descrittore di domande.

Ad esempio, il descrittore 1. specifica l'insieme costituito dall'unica domanda:

'NOME='

La risposta a tale domanda sarà allora interpretata come il valore da attribuire alla variabile "nome" (indicata nel descrittore della domanda stessa).
Così, in 4., se nome-cliente ha valore 'ROSSI', la domanda sarà:

'QUAL È L'INDIRIZZO DEL CLIENTE ROSSI?'

La risposta a tale domanda dovrà allora consistere nell'attribuzione di un valore alle tre variabili di nome "via", "numero", e "città".

c) - Descrittori di risposte

Per comprendere la notazione proposta per i descrittori di risposte, occorre ricordare la natura asimmetrica della classe di dialoghi che stiamo esaminando: un interlocutore (sistema) pone domande, e l'altro interlocutore(terminalista) risponde. Il dialogo procederà poi con una nuova domanda, selezionata dal primo interlocutore possibilmente in funzione del tipo di risposta ottenuta.
La notazione per i descrittori di risposte sarà pertanto scelta in modo tale da permettere di distinguere le diverse classi di risposte possibili a una stessa domanda, classi che determineranno le diverse successive evoluzioni del dialogo.

Esempi di descrittori di risposte (accostate ai descrittori delle domande relative, per maggiore chiarezza) sono :

1. domanda	:	'NOME=',<nome>
risposta	:	nome
2. domanda	:	'IL CLIENTE HA SALDATO IL CONTO?',<risposta>
risposta 1	:	risposta = 'NO'
risposta 2	:	risposta = 'SI'
risposta 3	:	ELSE
3. domanda	:	'DARE NOME E INDIRIZZO DEL CLIENTE', <nome>,<cognome>,<via>, <numero>,<città>
risposta 1	:	nome,cognome,via,numero,città='MILANO'
risposta 2	:	ELSE
4. domanda	:	'CREDITO='<credito>, 'DEBITO=',<debito>
risposta 1	:	credito ≥ debito
risposta 2	:	ELSE
5. domanda	:	'IL CLIENTE',nome(record(chiave)), 'HA SALDATO IL CONTO?', <risposta>
risposta 1	:	(risposta='NO' e fido (chiave) ≥ debito) o risposta='SI'
risposta 2	:	ELSE

Come si vede, un descrittore di risposte è costituito dai soliti elementi sintattici (tranne i nomi di variabili fra parentesi acute: messaggi costanti, nomi di variabili, nomi di funzioni), ed ha sempre la forma (esplicita o implicita) di una espressione logica.

Per chiarire il significato della notazione, esaminiamo gli esempi indicati.
La risposta alla domanda 1, è semplicemente: "nome". Ciò significa che l'interlocutore dovrà rispondere specificando il valore della variabile "nome" (la stessa racchiusa fra parentesi acute nella domanda relativa).
Per la domanda 2., invece, sono evidenziate tre diverse classi di risposte possibili, che determineranno in modo diverso l'evoluzione successiva del dialogo: una prima classe è costituita dall'unico valore 'NO' (che verrà attribuito alla variabile"risposta"), la seconda classe dall'unico valore 'SI', la terza classe è costituita da tutti gli altri possibili valori, indicati sinteticamen-

te per mezzo della parola chiave 'ELSE'. Così, la domanda 3. ha associate due diverse classi di risposte: la prima contiene tutte le risposte in cui nome, cognome, via e numero sono arbitrari e città è 'MILANO', la seconda in cui nome, cognome, via e numero sono arbitrari, e città è diversa da 'MILANO'.
In 4., la individuazione delle due classi di risposte è effettuata in base al valore relativo di credito e debito, mentre in 5. essa è più complessa, e coinvolge la valutazione di un'espressione logica in cui compaiono operatori logici (e, o), nomi di funzioni (fido) e altre variabili (chiave e debito) oltre a quelle specificate fra parentesi acute nel descrittore di domanda.

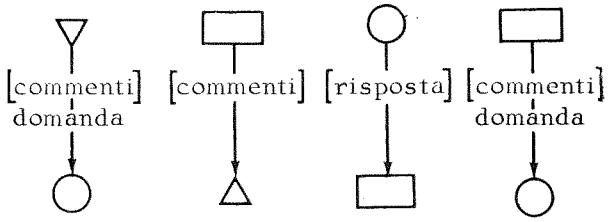
4. SCHEMI DI DIALOGO-DEFINIZIONE

Uno schema di dialogo è composto mediante i seguenti simboli:

- ▭ rappresenta uno "stato" dell'interlocutore-sistema
- ○ rappresenta uno "stato" dell'interlocutore-terminalista
- msg ↓ rappresenta una classe di messaggi che fanno passare da uno stato all'altro
- ▽ rappresenta lo stato iniziale (interlocutore-sistema)
- △ rappresenta lo stato finale (interlocutore-terminalista).

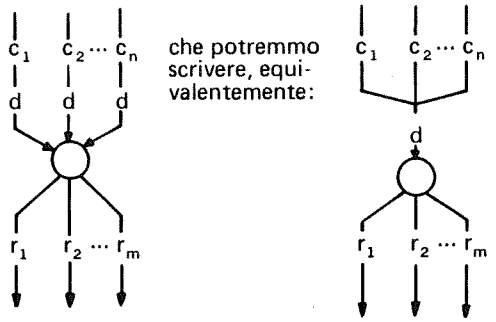
Le regole mediante le quali è possibile comporre uno schema di dialogo a partire da tali simboli sono le seguenti:

1. Ogni freccia connette due simboli di stato (▭ , ○ , ▽ , △).
Le connessioni possibili (che dipendono dalla natura dei messaggi) sono le seguenti:



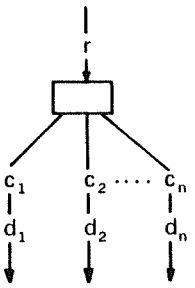
2. In ogni ○ entrano una o più frecce; in questo secondo caso, esse devono essere etichettate tutte con lo stesso descrittore di domanda (i commenti possono essere invece diversi, e possono essere vuoti).
Da ogni ○ escono una o più frecce: in questo secondo caso devono essere etichettate con descrittori di classi disgiunte di risposte.

Esempio:



3. In ogni ▭ entra una e una sola freccia; da ogni ▭ escono una o più frecce:

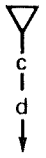
Esempio:



significa che l'interlocutore-sistema, ottenuta la risposta r, può porre o la domanda d_1 (preceduta dal commento c_1) o la domanda d_2 (preceduta dal commento c_2), o ... o la domanda d_n (preceduta dal commento c_n).

4. Esiste un solo ▽ : in esso non entra alcuna freccia, e da esso esce una sola freccia.

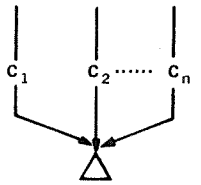
Esempio:



significa che il dialogo inizia con il commento c, seguito dalla prima domanda d.

5. Esiste al più un solo △ : in esso entrano una o più frecce, e da esso non esce alcuna freccia.

Esempio:



significa che il dialogo terminà con il commento c_1 o con il commento c_2 o ..... con il commento c_n, a seconda della sua storia.

Si noti che ammettiamo la possibilità di dialoghi che non terminano mai (cioè senza △: cfr. § 6).

5. ASSERZIONI

Così come in un programma, uno schema di dialogo può essere corredato da asserzioni. Queste saranno delle espressioni logiche associate ai vari rami dello schema, e coinvolgeranno costanti, variabili e funzioni non necessariamente presenti nei descrittori di messaggi.

Un esempio è mostrato in fig. 2, che mostra un frammento dello schema di fig.1 (modificato, per quanto riguarda i descrittori di messaggi, in accordo allo standard fornito nel paragrafo 3.

Si noti che la costante 'MELONE' e le due funzioni disponibilità(...) e disponibilità-di-magazzino (....) non compaiono nei descrittori di messaggi.
Naturalmente, ove un'asserzione sia inserita in un ciclo, essa dovrà costituire un invariante del dialogo, cioè dovrà risultare verificata ad ogni esecuzione del ciclo.

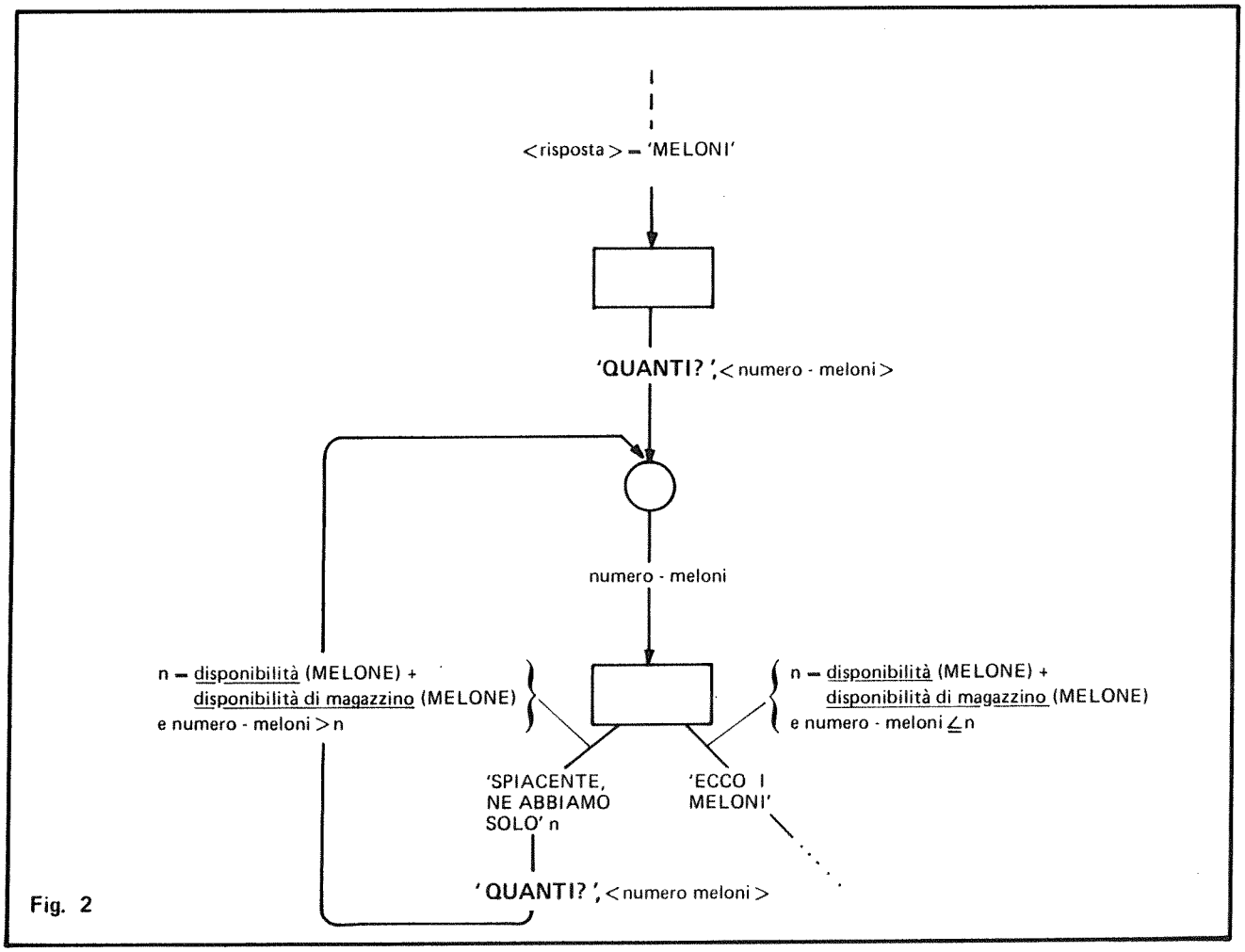


Fig. 2

6. UN ESEMPIO

In fig.3, 4 e 5 è mostrato un esempio completo di schema di dialogo per una prenotazione di posti aerei.
Si noti che lo schema di dialogo è presentato per raffinamenti successivi: α e β sono due sotto-schemi, ciascuno con un solo "ingresso" e una sola "uscita". Si noti, inoltre, che lo schema non termina: dopo ogni prenotazione, il sistema ripropone la domanda iniziale ('VOLO NAZIONALE?', <risposta>).

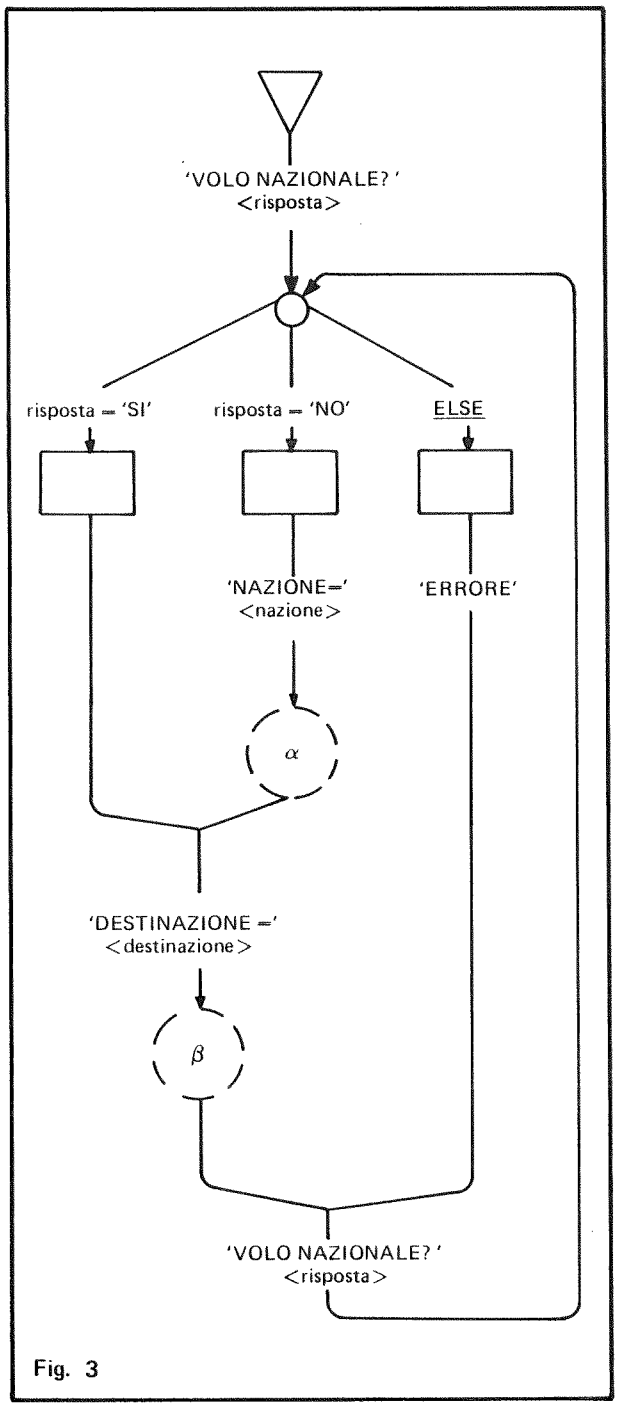


Fig. 3

7. INDIVIDUAZIONE DI STRUTTURE DI DIALOGO "TIPICHE"

Il linguaggio grafico introdotto permette di comporre schemi anche assai complessi. E' possibile, tuttavia, identificare nell'uso alcune strutture di dialogo relativamente semplici, che ricorrono molto frequentemente. Queste, se individuate in uno schema, ne forniscono, per così dire, una "chiave di lettura" e, soprattutto, permettono di disegnarlo in modo ordinato. Non è negli scopi di questa nota proporre una classificazione delle strutture di dialogo, motivata da criteri rigorosi. Ci limiteremo, pertanto, ad indicare una famiglia di strutture che, dall'analisi di numerosi esempi di applicazioni transazionali (che qui non riportiamo) sembra essere quella di gran lunga più ricor-

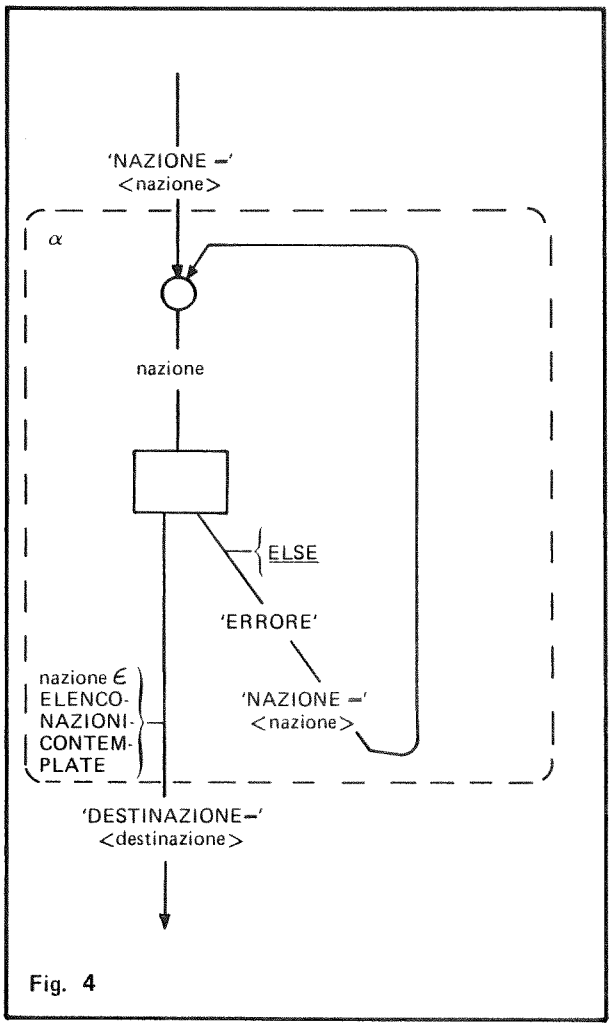


Fig. 4

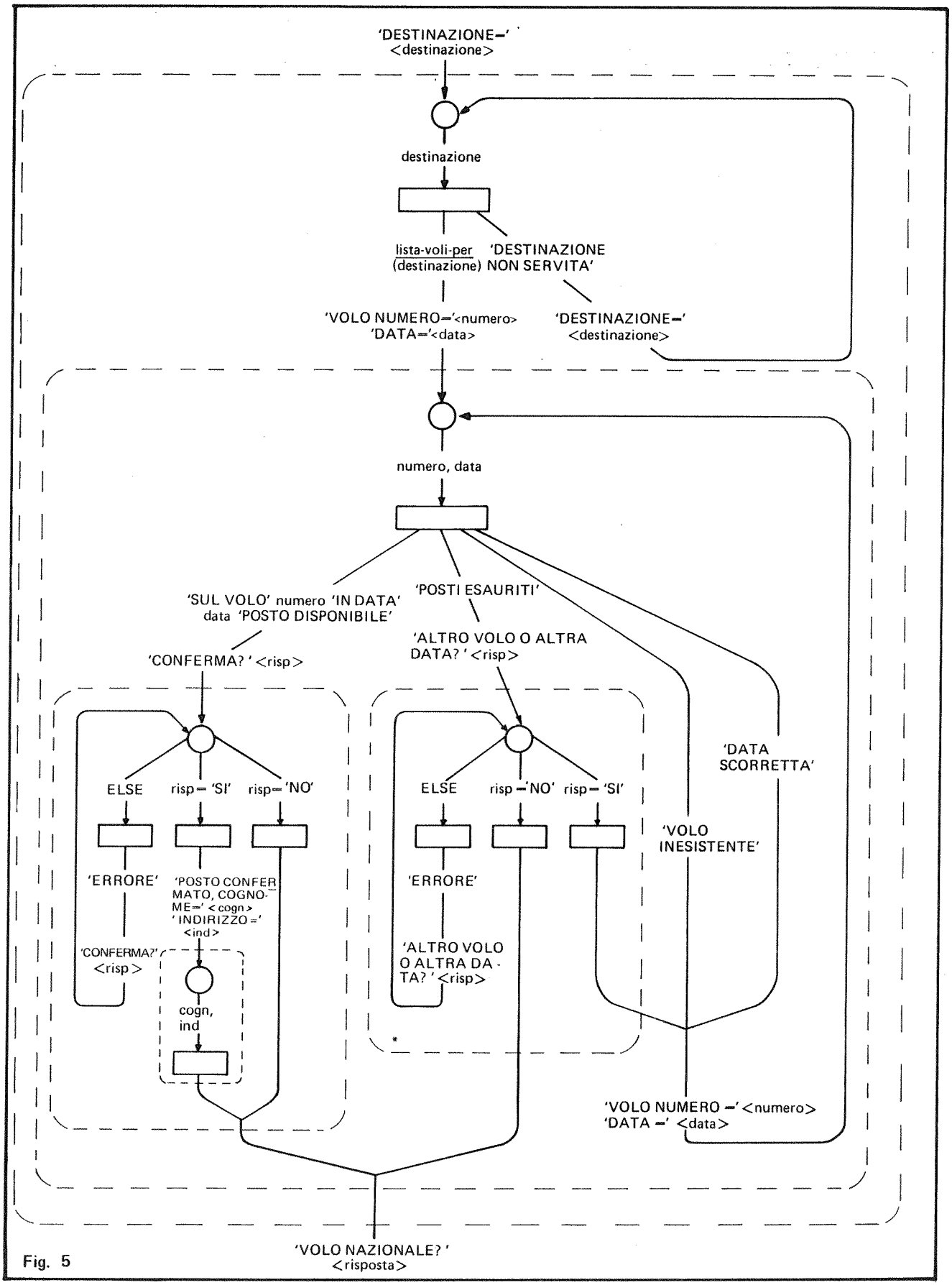

'DESTINAZIONE–' <destinazione>
destinazione
lista-voli-per (destinazione)
'DESTINAZIONE NON SERVITA'
'VOLO NUMERO–'<numero> 'DATA–'<data>
'DESTINAZIONE–' <destinazione>
numero, data
'SUL VOLO' numero 'IN DATA' data 'POSTO DISPONIBILE'
'POSTI ESAURITI'
'CONFERMA? ' <risp>
'ALTRO VOLO O ALTRA DATA? ' <risp>
ELSE
risp = 'SI'
risp = 'NO'
'ERRORE'
'POSTO CONFERMATO, COGNOME=' <cogn> 'INDIRIZZO=' <ind>
'CONFERMA?' <risp>
cogn, ind
ELSE
risp = 'NO'
risp = 'SI'
'ERRORE'
'ALTRO VOLO O ALTRA DATA? ' <risp>
'DATA SCORRETTA'
'VOLO INESISTENTE'
'VOLO NUMERO –' <numero> 'DATA –' <data>
'VOLO NAZIONALE? ' <risposta>

Fig. 5

rente. Queste strutture, particolarmente interessanti poichè hanno un solo ingresso e una sola uscita (permettendo pertando di rappresentare facilmente uno schema per livelli di raffinamento successivi, dal livello generale via via fino ai livelli più dettagliati) sono mostrate sinteticamente in fig.6, dove:

. i rami che escono da ◯ sono n, con $n \geq 1$;
. i rami che escono da ogni ▭ si ripartiscono in due classi:
.. quelli che alla fine si uniscono in un ramo di ritorno a ◯;
.. quelli che alla fine si uniscono nel ramo di uscita dalla struttura.
Ognuna delle due classi può essere vuota, o contenere un numero arbitrario di rami;
. ogni ● può essere sostituito a sua volta da una struttura della stessa classe, o da una semplice freccia, o da più strutture della stessa classe, concatenate in sequenza.

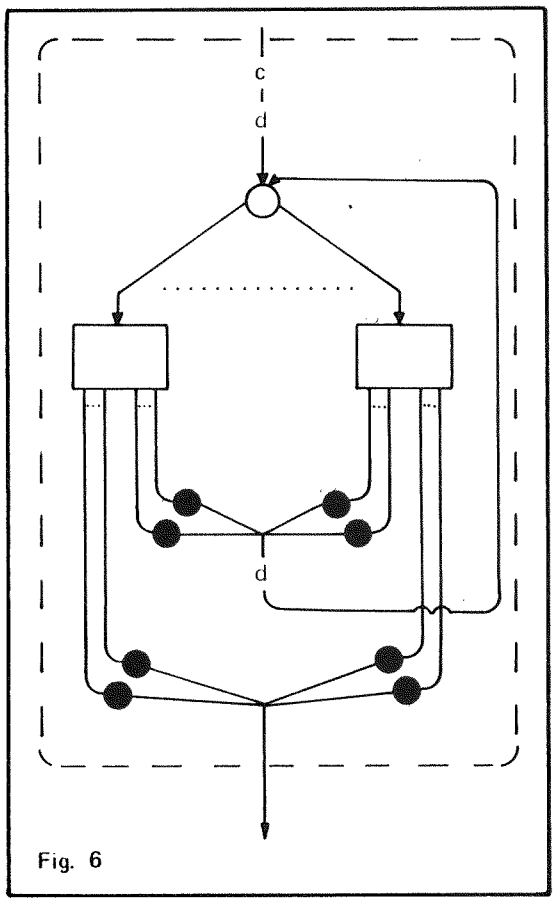
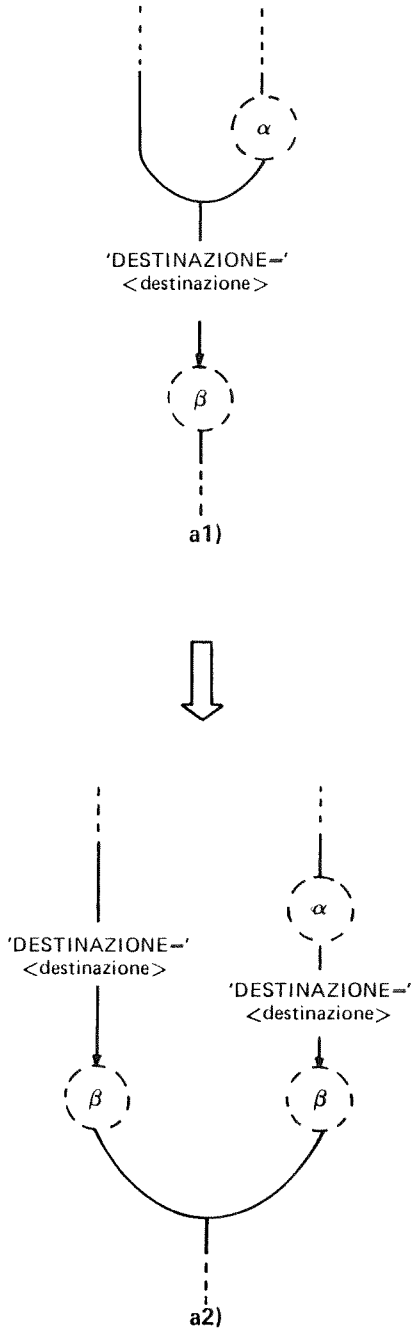


Fig. 6

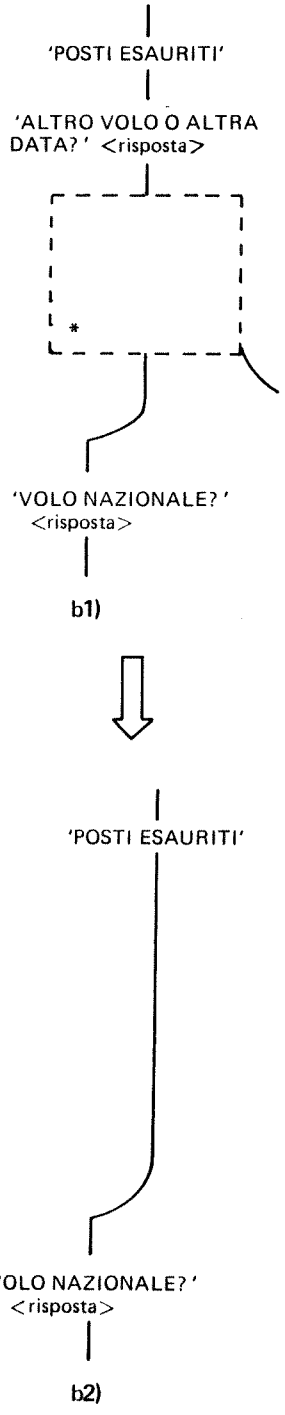
Casi particolari in questa famiglia di strutture sono riportati in fig. 7.

Ad esempio, consideriamo lo schema del paragrafo 6 (figg.3,4,5). Se esso viene modificato come segue:

a) - duplicando il blocco β in fig.3 e trasferendo ogni duplicato sui due archi provenienti da "risposta='SI'" e "risposta='NO'":



b) - sostituendo il sottodialogo successivo al commento 'POSTI ESAURITI' con una semplice freccia alla domanda 'VOLO NAZIONALE?':



allora, lo schema risultante è composto esclusivamente di strutture della famiglia descritta in fig.6.

Un'analisi delle due "anomalie" riscontrate conferma comunque, seppure su basi esclusivamente intuitive, l'interesse della famiglia di fig. 6. Infatti, la loro introduzione nello schema "modificato" secondo i punti a) e b) qui sopra può essere motivata da considerazioni di ottimizzazione:

a) - le due versioni a1) e a2) descrivono lo stesso insieme di dialoghi, ma a2) comporta duplicazione di un sotto-schema (e quindi del codice che lo realizza);

b) - le due versioni b1) e b2) descrivono due diversi insiemi di dialogo: b2) richiede che, nel caso di 'POSTI ESAURITI', per prenotare un volo alternativo il terminalista ripeta dall'inizio tutto il dialogo di prenotazione, mentre b1) risparmia la sequenza iniziale, ripetendo solo la domanda 'VOLO NUMERO=', 'DATA='.

7. SCHEMI DI DIALOGO E PROGRAMMI CHE LI REALIZZANO: ALCUNE OSSERVAZIONI

La struttura di uno schema di dialogo fornisce indicazioni precise sulla struttura del codice da implementare.
Possiamo osservare, ad esempio, che se il linguaggio di programmazione usato è di tipo strutturato, e dotato di una coppia di primitive send e receive (relative a uno specifico terminale), la struttura di fig.7c potrà essere, molto semplicemente, "tradotta" come:

```
send d
receive
do until r1
   elabora r2
   send c2
   send d
repeat
elabora r1
send c1
```

mentre lo schema di fig.7a sarà, più semplicemente

```
send d
receive
elabora r
send c
```

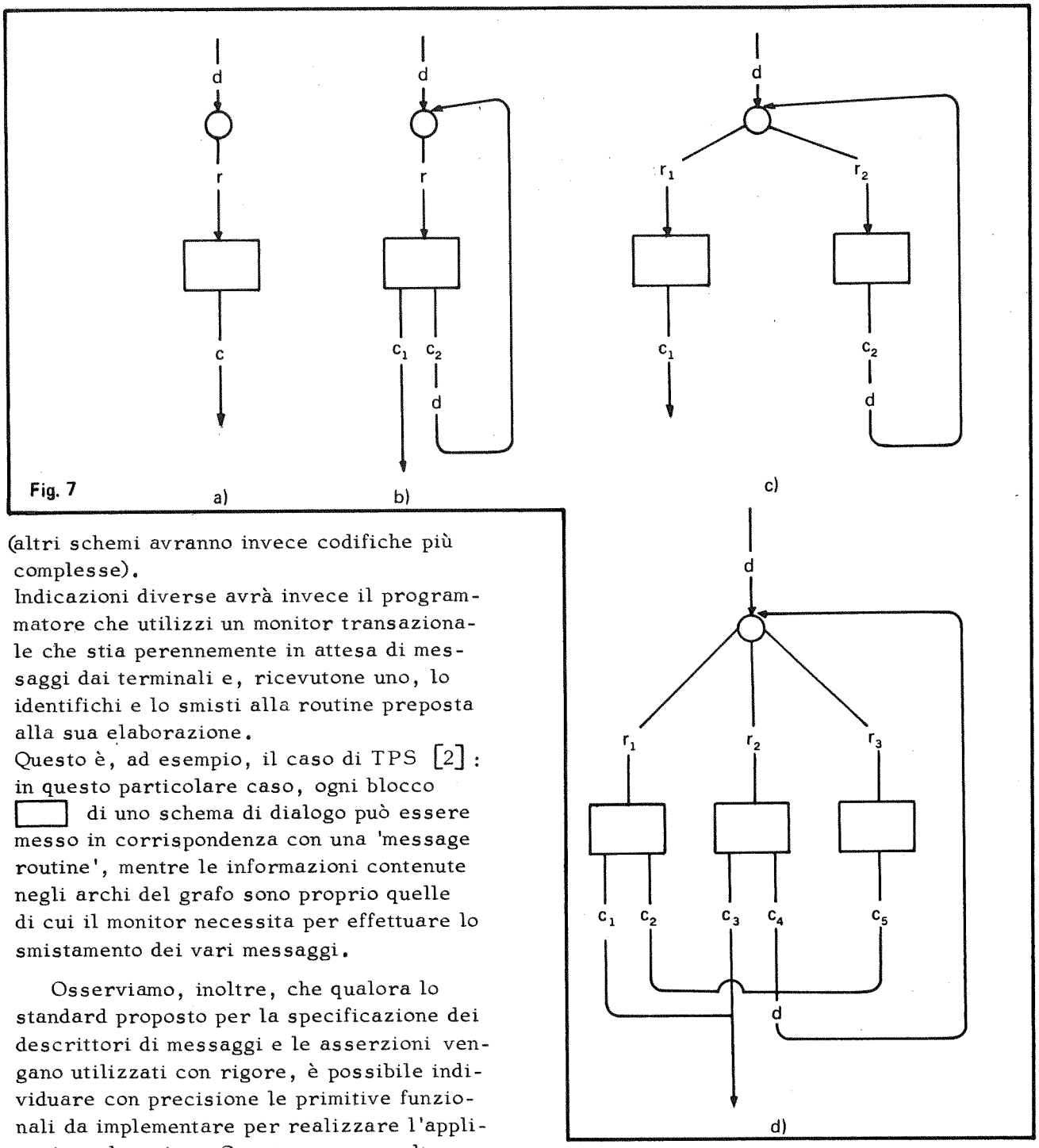


Fig. 7

(altri schemi avranno invece codifiche più complesse).
Indicazioni diverse avrà invece il programmatore che utilizzi un monitor transazionale che stia perennemente in attesa di messaggi dai terminali e, ricevutone uno, lo identifichi e lo smisti alla routine preposta alla sua elaborazione.
Questo è, ad esempio, il caso di TPS [2]: in questo particolare caso, ogni blocco ▭ di uno schema di dialogo può essere messo in corrispondenza con una 'message routine', mentre le informazioni contenute negli archi del grafo sono proprio quelle di cui il monitor necessita per effettuare lo smistamento dei vari messaggi.

Osserviamo, inoltre, che qualora lo standard proposto per la specificazione dei descrittori di messaggi e le asserzioni vengano utilizzati con rigore, è possibile individuare con precisione le primitive funzionali da implementare per realizzare l'applicazione descritta. Queste non sono altro che le funzioni e le relazioni che compaiono nei descrittori e nelle asserzioni stesse. (Ad esempio, in fig.2, le funzioni disponibilità(MELONE) e disponibilità-di-magazzino(MELONE)).

8. CONCLUSIONI

Nella presente nota è stata illustrata, con esempi, una semplice notazione per la descrizione dei dialoghi in un sistema interattivo. La notazione proposta da Denert [1] è stata qui particolarizzata tenendo conto di una specifica classe di applicazioni (applicazioni transazionali).

La notazione, che è stata sperimentata in una varietà di esempi (cfr. ad es. [3]) si rivela particolarmente efficace, soprattutto se utilizzata nella fase iniziale di progetta-

zione di un'applicazione transazionale (specifiche funzionali). Infatti:

- permette di descrivere in modo compatto e standardizzato, anche per successivi livelli di dettaglio, la visibilità completa che l'utente-terminalista avrà dell'applicazione in via di sviluppo, compresi gli aspetti di gestione di errori (o situazioni eccezionali);
- di conseguenza, permette di verificare, fin dalle primissime fasi del progetto, la operabilità del sistema (ad esempio mediante simulazioni manuali dello sviluppo del dialogo);
- fornisce indicazioni precise sulla struttura del codice da implementare.

9. RIFERIMENTI

[1] E. Denert, SPECIFICATION AND DESIGN OF DIALOGUE SYSTEMS WITH STATE DIAGRAMS, Proceedings of the International Computing Symposium 1977 (preprints), Liège, 4-7 aprile 1977 (E. Morlet, D.Ribbens editors.) pagg.417-424.

[2] TPS SYSTEM MANUAL, H.I.S.I., order number: AY72, Rev. 1, dicembre 1977.

[3] S. Cappelli, La gestione dei flussi informativi in un sistema organizzato: un modello, Univ. di Milano, tesi di laurea in filosofia, in preparazione.

PHOS: una metodologia per la costruzione veloce ed affidabile di programmi

T. Bettinazzi, [^+] M. Maiocchi, [^+] A. Maserati, [^+] F. Monzani, [^+] E. Spoletini, [^] E. Toscani [^+]

1. Introduzione

Lo scopo delle metodologie di programmazione, il cui uso, in particolare in ambienti EDP, da qualche tempo è sempre più frequente, è quello di fornire ai programmatori strumenti e linee guida sicure che consentano di ottenere programmi di buona qualità cioè privi di errori e facilmente manutenibili (cfr., ad es., 1,2). Inoltre, si sono precisati altri obiettivi: una metodologia deve fornire uno strumento di standardizzazione di una cultura tecnica, così da rendere i programmi comunicabili e l'esperienza trasferibile da programmatore a programmatore; inoltre, deve garantire che la qualità di un programma si conservi nel tempo, nonostante gli interventi di manutenzione.

Gli strumenti che vengono forniti dalle metodologie riducono l'importanza di aspetti di sofisticata conoscenza tecnica dei linguaggi di programmazione e dei calcolatori, così da poter concentrare l'attenzione su aspetti legati più tipicamente al problema da risolvere, orientando così l'attività di costruzione dei programmi verso l'end - user, a cui non viene richiesto un livello di preparazione tecnica eccessivamente elevato.

In tali ottiche, una metodologia di programmazione deve:

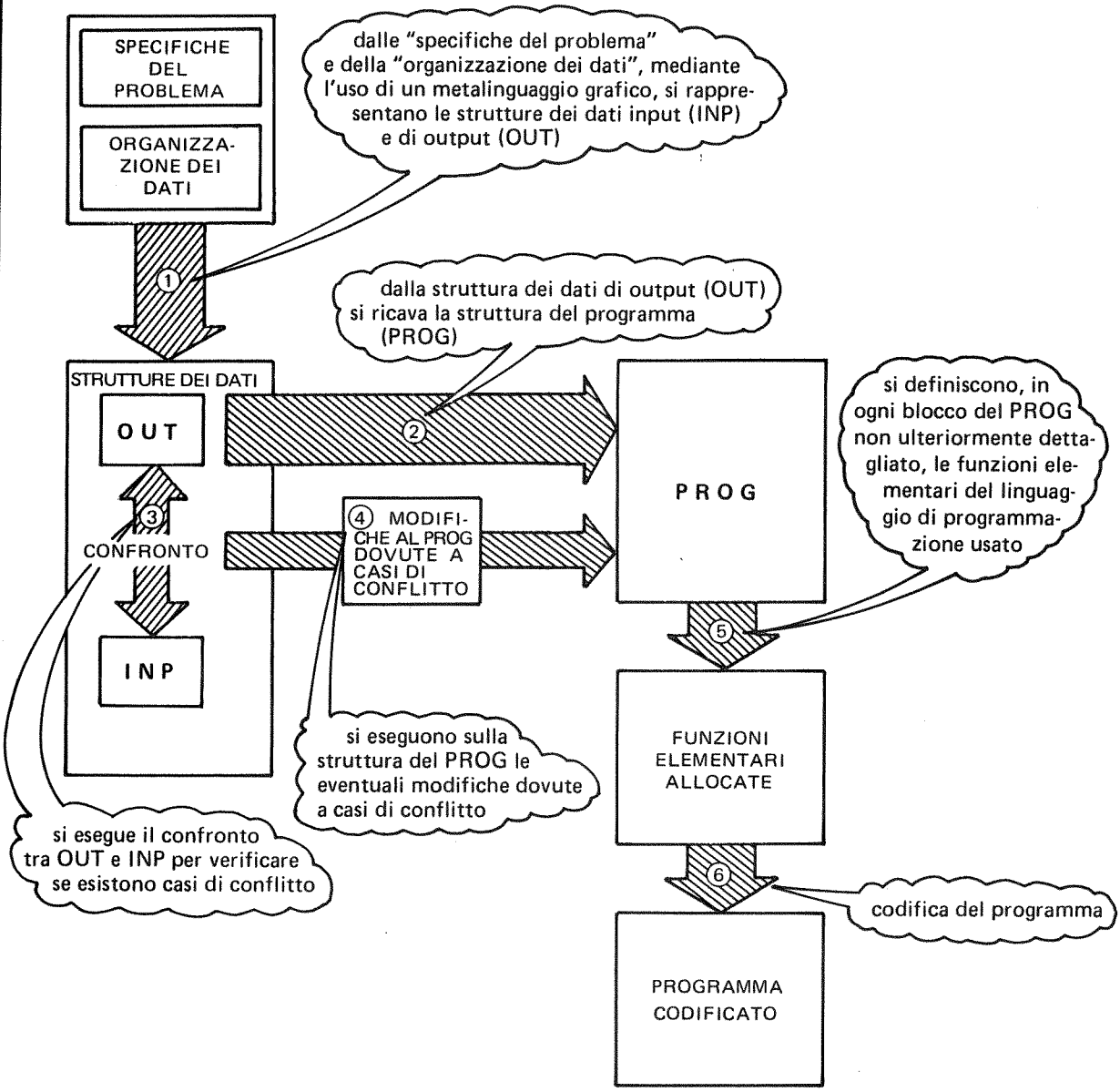
- fornire strumenti concettuali per effettuare una valida analisi critica delle specifiche del programma da costruire
- fornire sicure indicazioni per tracciare la struttura architetturale del programma a partire dalle specifiche
- mettere a disposizione, per effettuare le operazioni descritte, opportuni divisori linguistici, che siano facili da imparare ed usare, intuitivi nell'interpretazione e contemporaneamente privi di ambiguità
- fornire standard di codifica che garantiscano uniformità nella costruzione dei programmi, qualità nella loro struttura fisica reale e garantiscano inoltre la conservazione nel tempo di tali caratteristiche.

Il presente articolo descrive la metodologia di programmazione PHOS (Program Hierarchy from Output Structure), che si indirizza esplicitamente agli scopi indicati, lasciando ad un successivo articolo il compito di mostrare una applicazione della metodologia su un problema reale.

La metodologia, che è stata sviluppata nell'ambito del Communication and Programming Project da un gruppo di lavoro composto da C. Barassi, T. Bettinazzi, D. Biglino, M. Maiocchi, A. Maserati, F. Monzani, G. Murè, E. Spoletini, E. Toscani, è indipendente da specifici linguaggi di programmazione e si è dimostrata particolarmente adatta per la risoluzione di problemi in ambiente EDP.
Gli autori ringraziano G. Degli Antoni per gli efficaci suggerimenti forniti durante lo sviluppo della metodologia.

[^] Honeywell Information Systems Italia
[+] Università di Milano - Istituto di Cibernetica

PASSI DELLA METODOLOGIA PHOS



La metodologia PHOS indica rigidamente le fasi da percorrere per la costruzione del programma: il risultato delle varie fasi, indicato in figura,é il programma codificato nel linguaggio di programmazione scelto insieme alla documentazione di disegno del programma stesso.

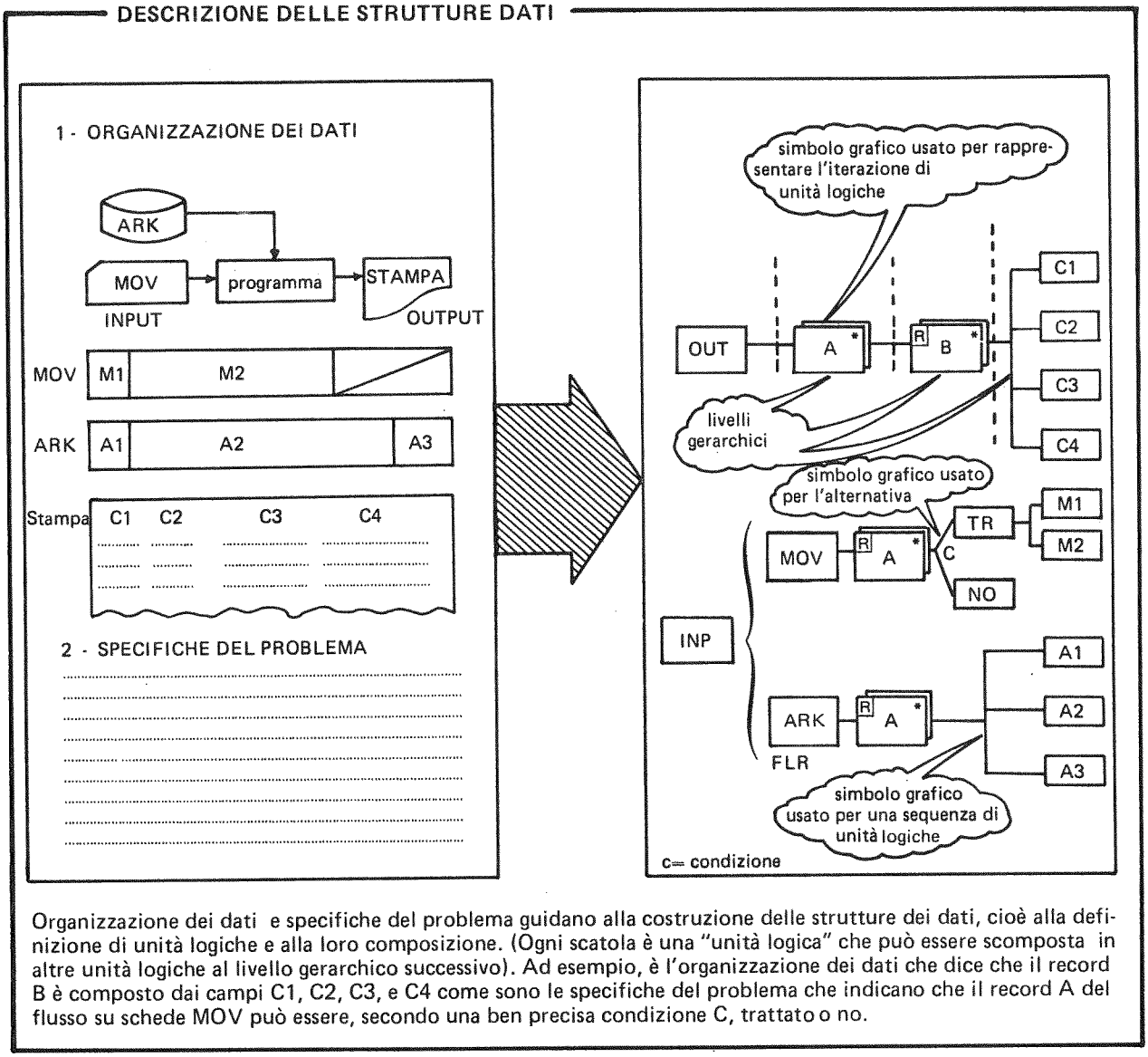
2. La metodologia PHOS

La metodologia individua un certo insieme di fasi di lavoro che il programmatore deve svolgere.
Le fasi sono opportunamente e rigidamente sequenziate; per ciascuna di esse sono indicate le "materie prime" su cui operare ed il "prodotto finito" risultante, e per ciascuna di esse sono fornite linee guida, piú o meno vincolate a seconda delle fasi, per ottenere i risultati necessari.

Opportuni strumenti linguistici accompagnano gli strumenti concettuali.

Le fasi in cui la metodologia articola lo sviluppo di un programma sono le seguenti:

a. descrizione gerarchica delle strutture dei dati, di ingresso e di uscita, coinvolte dal programma
b. derivazione della struttura del programma a partire dalla struttura dei dati di uscita
c. analisi di eventuali situazioni di contrasto di struttura tra dati di ingresso e dati di uscita; studio e risoluzione delle situazioni di conflitto che impongono modifiche alla struttura del programma precedentemente costruita
d. inserimento delle operazioni elementari (letture, calcoli, stampe, ecc.) nella struttura del programma
e. codifica del programma.



Organizzazione dei dati e specifiche del problema guidano alla costruzione delle strutture dei dati, cioè alla definizione di unità logiche e alla loro composizione. (Ogni scatola è una "unità logica" che può essere scomposta in altre unità logiche al livello gerarchico successivo). Ad esempio, è l'organizzazione dei dati che dice che il record B è composto dai campi C1, C2, C3, e C4 come sono le specifiche del problema che indicano che il record A del flusso su schede MOV può essere, secondo una ben precisa condizione C, trattato o no.

3. Descrizione delle strutture di dati

I dati vengono descritti in una struttura gerarchica che utilizza solo tre regole standard di composizione: sequenza, alternativa, plurale.

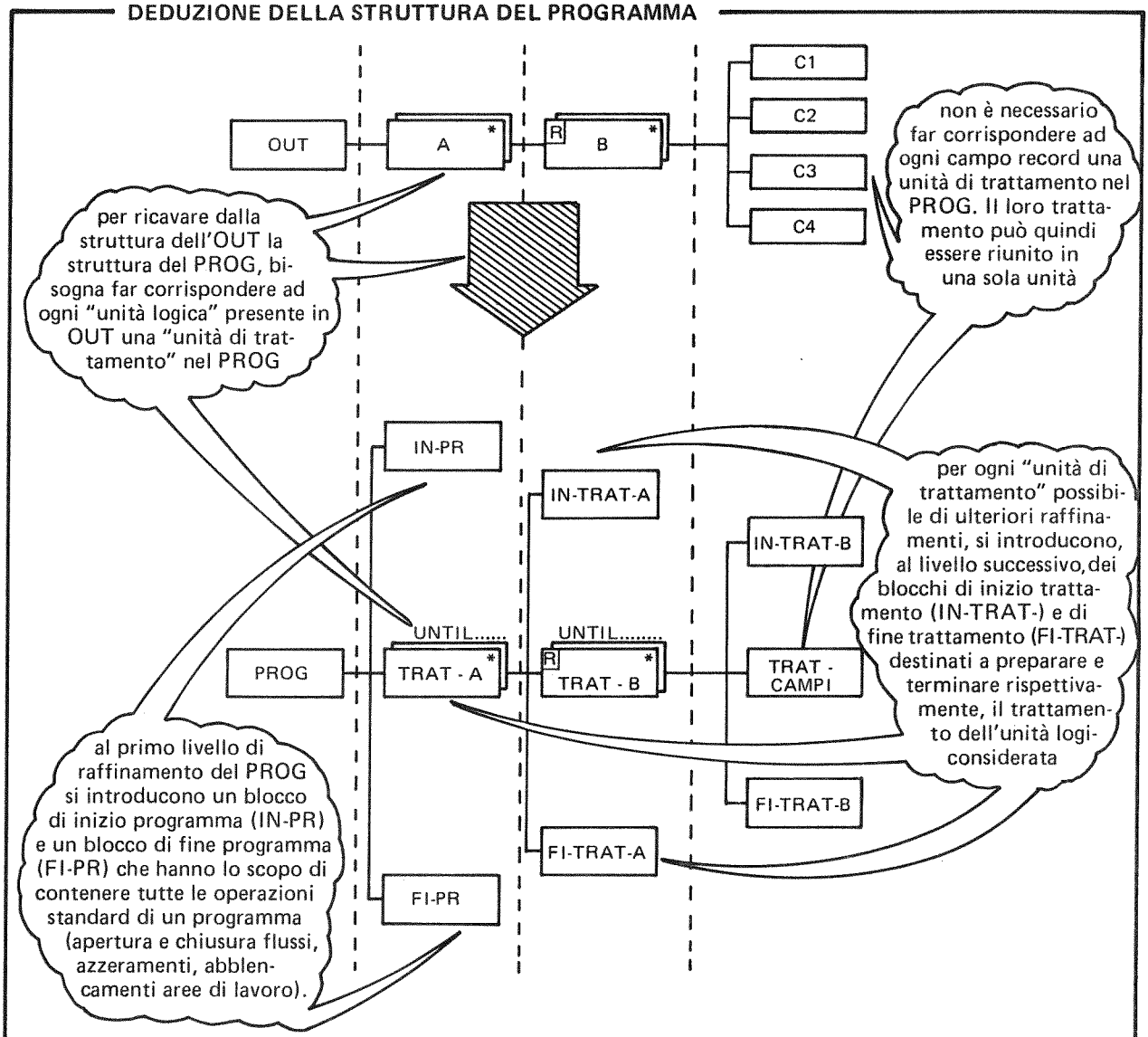
L'obiettivo di tale descrizione gerarchica è quello di individuare le entità concettualmente rilevanti nel problema, cioè le "unità logiche", e di mostrare come queste "unità logiche" sono fra loro in relazione.

Così, una "unità logica" A potrà essere definita come:

- una sequenza di "unità logiche" B, C, ..., D di livello immediatamente inferiore,
- un'alternativa fra due "unità logiche" B e C di livello immediatamente inferiore (cioè A è o B o C, sulla base di una certa condizione),
- una iterazione di più "unità logiche" dello stesso tipo, B (cioè A è costituito da più B).

Tali unità logiche, descritte graficamente con una notazione opportuna, possono essere considerate gli elementi di una "grammatica" strutturale dei dati di ingresso e di uscita.

In particolare, allo scopo di individuare unità logiche i cui componenti debbano essere sottoposti ad un trattamento uniforme, si impone che una sola delle tre regole di composizione possa essere usata ad ogni livello della descrizione gerarchica.



Nella costruzione della struttura del programma viene impiegato lo stesso formalismo della descrizione dei dati. Si osservi come la condizione di uscita dell'iterazione (UNTIL . . . . .) sia direttamente indicata sullo schema, al fine di renderlo autoconsistente al lettore.

4. Deduzione della struttura del programma

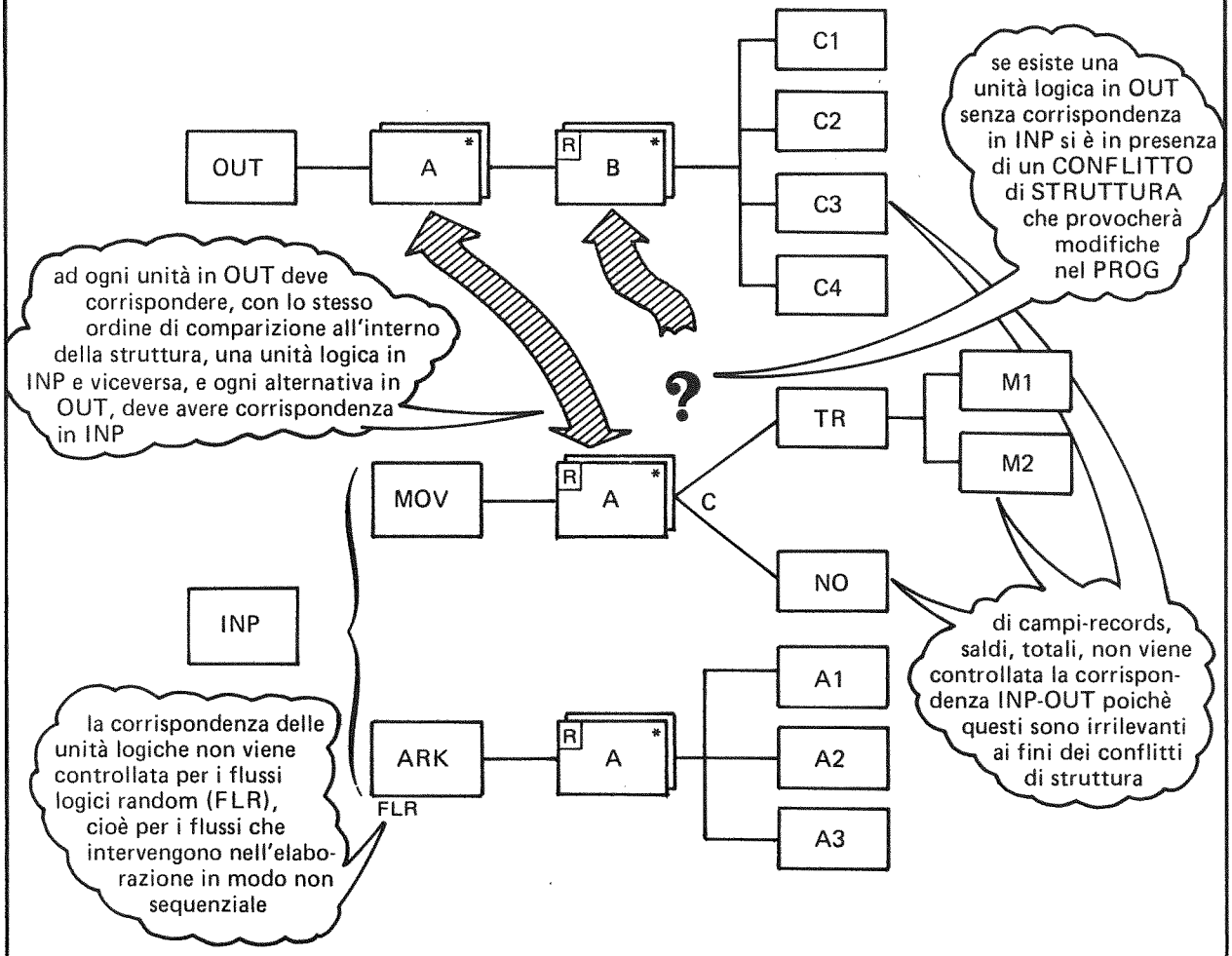
Un programma che legga successioni di informazioni d'ingresso e produce sequenze di risultati può essere rozzamente confrontato con una grammatica rivolta al riconoscimento di dati (nel caso si ponga l'accento sulla produzione di risultati). Pertanto si può pensare che sia necessario fornire al programma una struttura completamente aderente alla struttura dei dati in ingresso, oppure, alternativamente, a quella dei dati in uscita.

Si è scelto, nella metodologia PHOS, di modellare la struttura del programma sulla base della struttura dei risultati, in quanto si è ritenuta più rilevante la funzione "produzione di risultati"; quindi i risultati vengono descritti in un'unica struttura gerarchica basata sulla loro "sequenza di generazione" nel tempo, e indipendente dai supporti utilizzati.

La struttura del programma, che viene definita semplicemente ricopiando la struttura dei risultati, risulterà completamente adeguata alle elaborazioni necessarie, infatti la struttura dei risultati è stata definita mediante l'individuazione di unità logiche, individuate sulla base delle specifiche del problema.

I dati d'ingresso vengono visti in un ruolo subordinato, e ne viene mantenuta una descrizione separata per i dati che risiedono su supporti fisici differenti.

VERIFICA DEI CONFLITTI DI STRUTTURA TRA OUT E INP



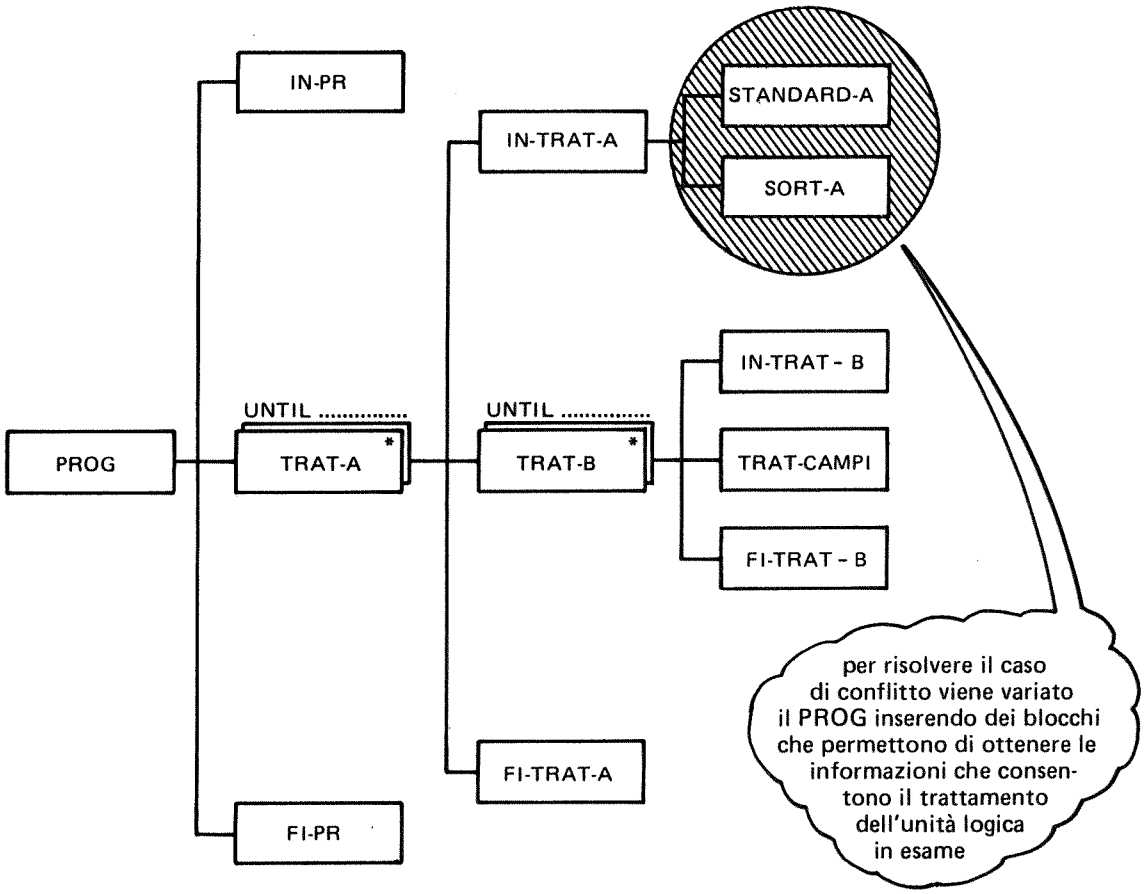
L'individuazione di conflitti avviene mediante ispezione, livello per livello, delle strutture dei dati di ingresso ed uscita. In figura è illustrato uno dei possibili casi di conflitto classificati nella metodologia.

5. Risoluzione di conflitti

Si può presentare il caso che un programma modellato come "generatore" di dati in uscita non sia adeguato come "riconoscitore" dei dati in ingresso, a causa di profonde differenze nella struttura logica dei due tipi di informazioni. In tal caso, è necessario apportare alla struttura del programma delle modifiche che eliminino l'inconveniente.

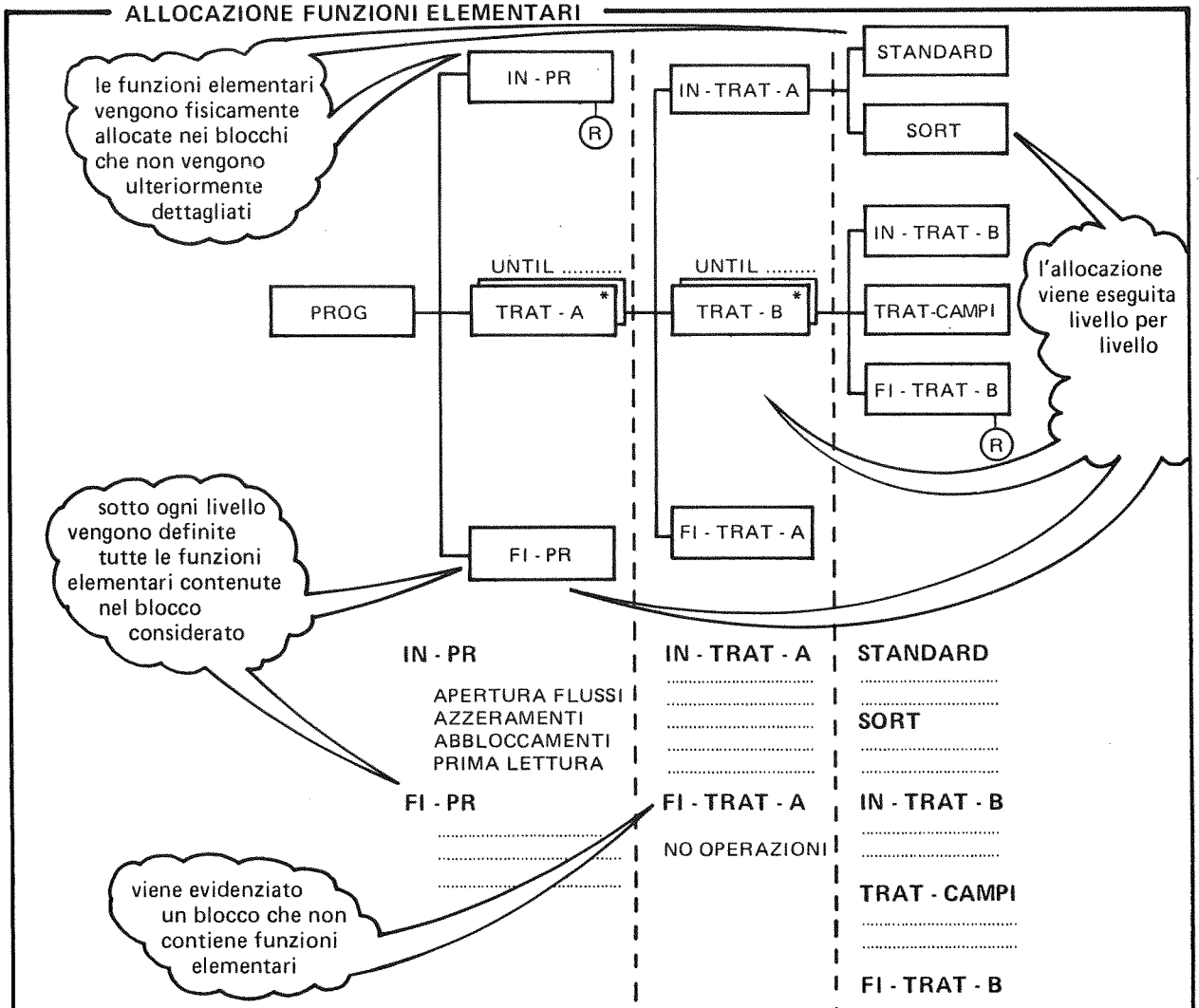
La metodologia fornisce opportuni strumenti per riconoscere le situazioni di conflitto; tali strumenti sono basati sull'ispezione delle strutture dei dati, e permettono la soluzione adeguata a seconda del tipo di conflitto.

MODIFICHE AL PROG DOVUTE A CASI DI CONFLITTO DI STRUTTURA TRA INP E OUT



Il conflitto precedente viene risolto semplicemente con l'inserimento di un ordinamento di dati (in altri casi si rivela necessario spezzare il programma in diversi programmi che comunicano tra di loro via files intermedi che non presentano i conflitti individuati).

Le trasformazioni imposte alla struttura del programma consistono, generalmente, nell'introduzione di opportune porzioni in modo standard, che hanno lo scopo di trasformare i dati di ingresso in una forma intermedia che non presenti conflitti con la struttura dei risultati.



N.B. - il livello delle funzioni elementari descritte dipende solo dal linguaggio considerato (nel nostro esempio COBOL)

Le funzioni elementari vengono allocate sulla base di precise regole: ad esempio "in ogni blocco iniziale di un programma intermedio devono essere inseriti:

- memorizzazioni di codici per interruzione di iterazione
- azzeramenti di aree locali
- preparazioni di stampe"

Indicazioni di tale tipo vengono fornite per blocchi iniziali e finali di livelli intermedi e per il blocco iniziale e finale del programma globale.

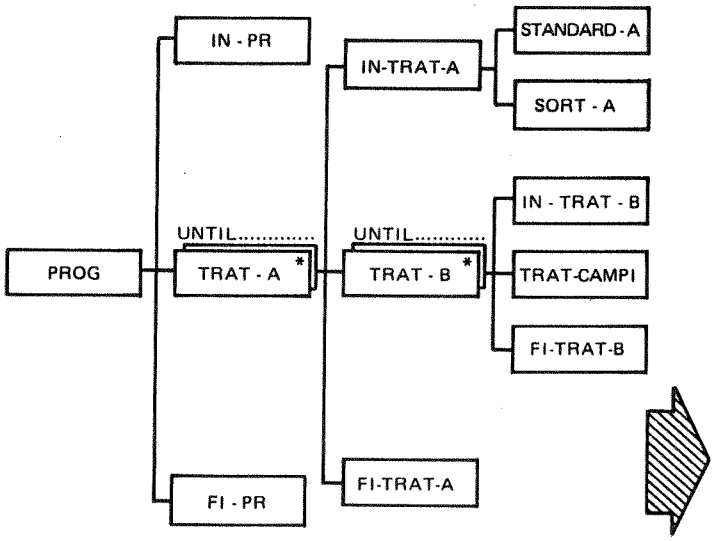
6. Allocazione delle funzioni elementari

Una volta individuata la struttura definitiva del programma, è necessario che le diverse porzioni individuate vengano "riempite" con le opportune istruzioni per l'effettuazione delle trasformazioni richieste.

Tali "funzioni elementari", che troveranno poi immediata corrispondenza con le istruzioni del linguaggio di programmazione utilizzato per la codifica, vengono inserite nella struttura in un modo rigorosamente top-down, sulla base di precise indicazioni.
Si tratta di regole che indicano quali operazioni devono essere previste ed inserite, e dove tali operazioni devono essere collocate.

In generale molte di esse, cioè tutte quelle destinate alla gestione dei files, alla costruzione di totali, alla gestione di iterazioni e selezioni, vengono inserite in blocchi iniziali e finali di ciascun livello del programma, secondo le regole fornite.

CODIFICA




```
PROCEDURE DIVISION.
MAIN SECTION.
ALBERO-PROGRAMMA.
**
*** PRIMO LIVELLO
*
    PERFORM  IN - PR
    PERFORM  TRAT - A  UNTIL  EOF = "SI"
    PERFORM  FI - PR
    STOP RUN.
*
*** SECONDO LIVELLO
*
TRAT - A.
    PERFORM  IN - TRAT - A
    PERFORM  TRAT - B UNTIL  COD NOT = DEP
    PERFORM  FI - TRAT -A.
*
*** TERZO LIVELLO
*
IN - TRAT - A.
    PERFORM  STANDARD - A
    PERFORM  SORT - A.
TRAT - B.
    PERFORM  IN - TRAT - B
    PERFORM  TRAT - CAMPI
    PERFORM  FI - TRAT - B.
*
*** ALLOCAZIONI FUNZIONI ELEMENTARI
*** PRIMO LIVELLO
*
IN - PR SECTION.
.....
.....
FI - PR SECTION.
.....
.....
.....
```

IN - PR
.....
.....
.....

FI - PR
.....
.....
.....
.....

IN - TRAT - A
.....
.....
.....

FI - TRAT - A
.....
.....
.....
.....

STANDARD
.....
.....

SORT
.....
.....

IN - TRAT - B
.....
.....

TRAT - CAMPI
.....
.....

IN - TRAT - B
.....
.....
.....
.....

Questo esempio di codifica esemplifica l'uso del Cobol, con la metodologia, che è completamente indipendente dal linguaggio di programmazione che interviene solo nell'ultima fase.

7. Codifica

Gli obiettivi delle indicazioni fornite per la codifica del programma sono:
rispettare nella struttura fisica della codifica la struttura logica del programma, ottenere un prodotto particolarmente leggibile e, sopratutto, ottenere un programma in cui tali caratteristiche sopravvivono ad ogni modifica.

Ciò richiede che il linguaggio di programmazione sia completamente adeguato a rappresentare gli strumenti di descrizione del programma precedentemente usato, in particolare le descrizioni gerarchiche dei dati, le strutture di controllo per la selezione e l'iterazione ad un solo ingresso ed una sola uscita e la articolazione gerarchica del programma.

La metodologia è stata completata, per quanto riguarda le indicazioni sulla codifica, facendo riferimento al linguaggio COBOL, di cui viene impiegato un sottoinsieme particolarmente semplice, in particolare per ciò che riguarda le strutture di controllo, che si limitano all'uso di istruzioni IF..... ELSE e di clausole PERFORM UNTIL. La PERFORM viene inoltre estensivamente usata, per conferire al programma la struttura gerarchica prodotta nell'analisi.

8. Sviluppi

Durante il 1977 presso il servizio Scuole Marketing della HISI sono stati fatti, utilizzando la metodologia PHOS, alcuni corsi sperimentali di formazione per programmatori. La metodologia ha dato ottimi risultati, sia per l'aspetto didattico sia per quello metodologico.
Nel corso del 1978 saranno gradualmente proposti corsi di programmazione secondo la metodologia PHOS, sia utilizzando la forma consueta di didattica in aula, sia con un training effettuato in autoistruzione e con workshop.

Attualmente, si sta sviluppando PHOS per renderla particolarmente adattabile a situazioni di programmazione che coinvolgono Tele Processing e Data Bases. Inoltre è in corso di messa a punto il completamento della metodologia per quanto riguarda il testing dei programmi costruiti.

9 Riferimenti

1) J. Warnier, B. M. Flanagan, "Entrainement à la programmation", 1972, Les Editions d'Organisation, Paris

2) M. Jackson, "Principles of program design", 1975, Academic Press Ldt

3) T. Bettinazzi, A. Maserati, F. Monzani, E. Toscani, "PHOS: Una Applicazione in Ambiente EDP", Note di Software N.4

P.H.O.S. : UNA APPLICAZIONE IN AMBIENTE E.D.P.

T. Bettinazzi [0+], A. Maserati [0+], F. Monzani [0+], E. Toscani [0+]

1. INTRODUZIONE

L'obiettivo del presente articolo è quello di mostrare una applicazione della metodologia PHOS fornendone, al contempo, i principi chiave ed una guida pratica al suo utilizzo. Tale metodologia è stata sviluppata nell'ambito del C. P. Project in collaborazione tra l'Istituto di Cibernetica dell'Università di Milano e la Honeywell I.S.I.

L'esempio propone l'applicazione della metodologia ad un tipico caso di gestione EDP, privo di particolari difficoltà concettuali ma esaustivo dal punto di vista didattico. Al lettore attento potranno apparire criticabili alcune scelte indicate nell'analisi (per esempio quella di ignorare completamente i clienti senza corrispettivo in ARK-CLI): tali scelte tuttavia sono state conservate al fine di calare in un esempio particolarmente semplice diversi aspetti di uso della metodologia.

Nel presente articolo non si entrerà nel merito degli aspetti concettuali propri della metodologia (1) (2); tuttavia, per rendere più chiara l'esposizione, sintetizziamo qui le linee guida fornite dal metodo e la sequenza della loro applicazione:

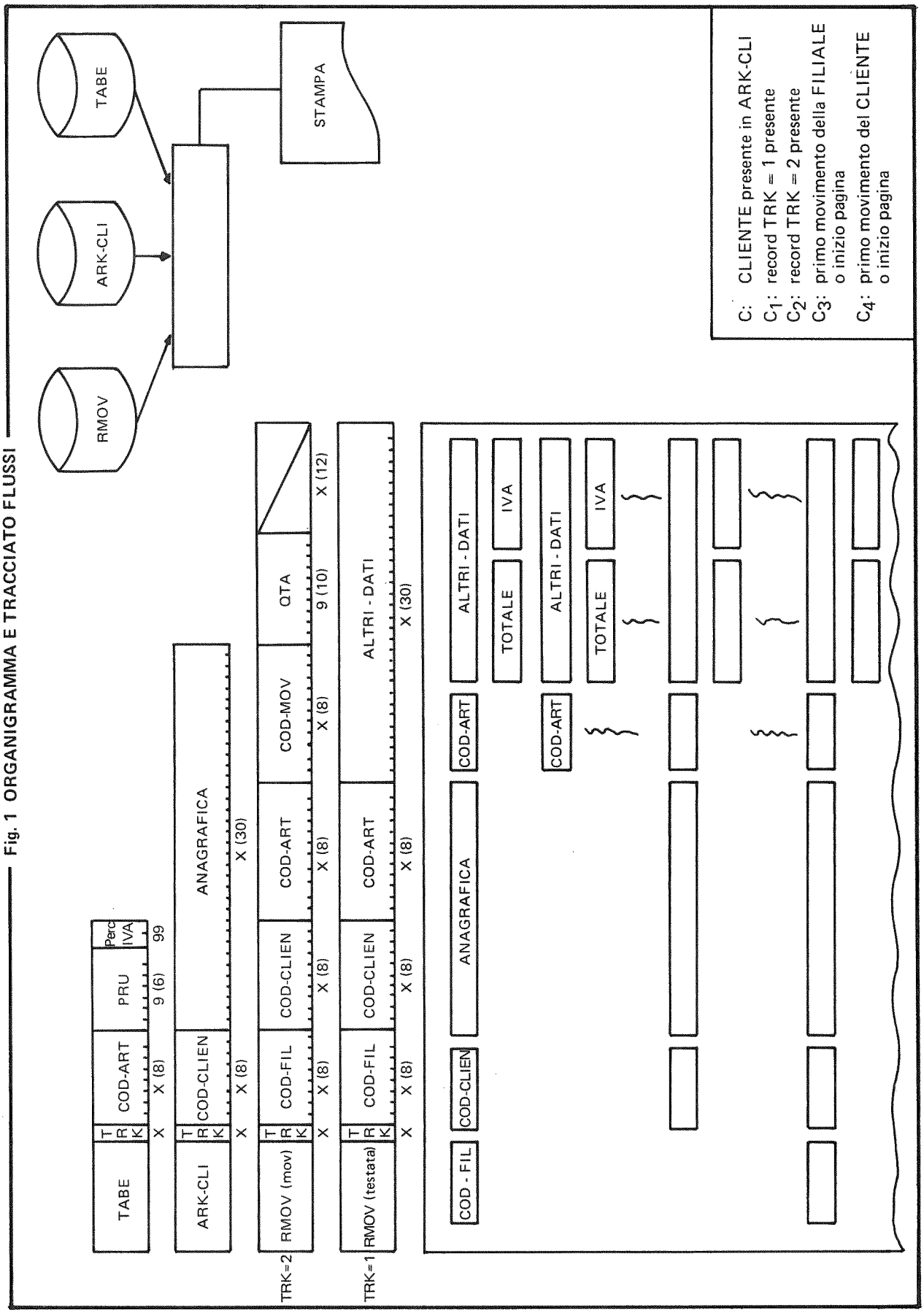
- 1° passo: individuazione della struttura dei dati a partire dalle specifiche del problema;
- 2° passo: derivazione della struttura del programma da quella dei dati di output;
- 3° passo: confronto delle strutture di input con quelle di output;
- 4° passo: modifica del programma per la risoluzione dei casi di conflitto;
- 5° passo: allocazione delle funzioni elementari;
- 6° passo: codifica standard in Cobol.

Il problema presentato, del quale vengono fornite di seguito le specifiche, verrà risolto sviluppando, punto per punto, i precedenti passi, a ciascuno dei quali verrà dedicato un singolo paragrafo.

[0] Istituto di Cibernetica dell'Università di Milano

[+] Honeywell I.S.I.

Fig. 1 ORGANIGRAMMA E TRACCIATO FLUSSI



2. SPECIFICHE DEL PROBLEMA

Le specifiche del problema trattato in questo esempio possono essere così sintetizzate:

INPUT:

- Flusso TABE, ad organizzazione sequenziale, da utilizzarsi per il caricamento di una tabella articoli, contenente i valori del prezzo unitario e della percentuale IVA.
- Flusso ARK-CLI, ad organizzazione indexed, da trattarsi in accesso random, contenente un record per ogni cliente, con i relativi dati anagrafici.
- Flusso RMOV, ad organizzazione sequenziale, ordinato su codice-filiale, codice-cliente, codice-articolo, contenente, per ogni articolo, un record testata e un gruppo di records movimento.

OUTPUT:

- Tabulato con valorizzazioni articoli, come da tracciato.
- Segnalazioni d'errore su consolle.

ELABORAZIONE:

- Caricare in memoria la tabella scorrendo sequenzialmente il flusso TABE.
- Scorrere sequenzialmente il flusso RMOV, totalizzando i movimenti solo per i clienti che presentano corrispettivo in ARK-CLI; clienti senza corrispettivo vanno ignorati.
- Per gli articoli sprovvisti di record movimento, è richiesta una segnalazione su consolle e non si deve procedere alla stampa.
- Per gli articoli sprovvisti di record testata, lasciare bianca la zona "altri dati" sul tabulato.
- Il codice-filiale va stampato solo per il primo articolo della filiale, il codice-cliente solo per il primo articolo del cliente; i codici vanno ripetuti a pagina nuova.
- Massimo 50 righe per foglio; non andare a pagina nuova per i soli totali.

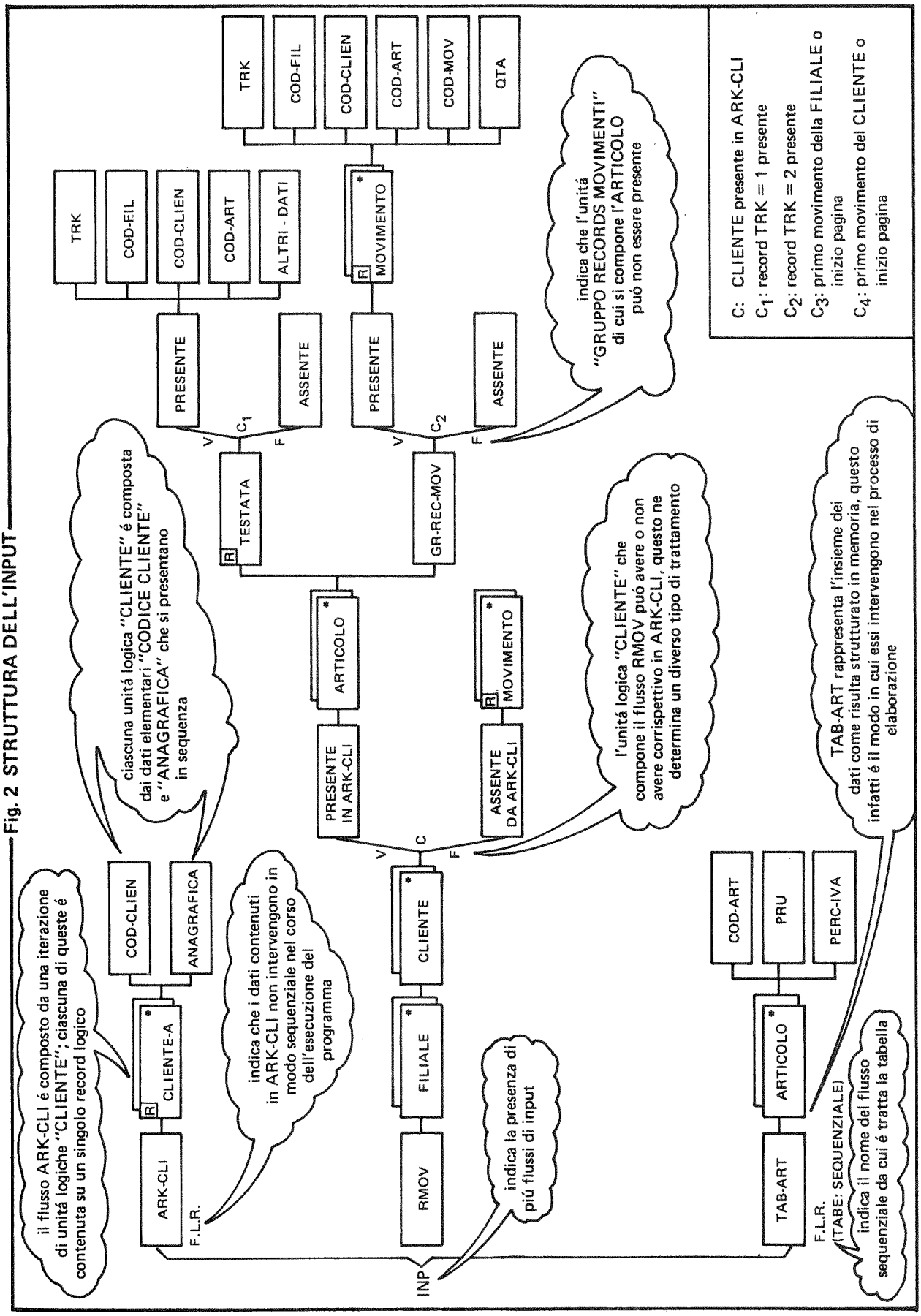


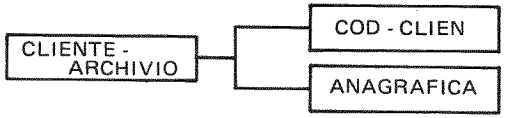
Fig. 2 STRUTTURA DELL'INPUT
il flusso ARK-CLI é composto da una iterazione di unitá logiche "CLIENTE"; ciascuna di queste é contenuta su un singolo record logico
ciascuna unitá logica "CLIENTE" é composta dai dati elementari "CODICE CLIENTE" e "ANAGRAFICA" che si presentano in sequenza
indica che i dati contenuti in ARK-CLI non intervengono in modo sequenziale nel corso dell'esecuzione del programma
indica la presenza di piú flussi di input
l'unitá logica "CLIENTE" che compone il flusso RMOV puó avere o non avere corrispettivo in ARK-CLI, questo ne determina un diverso tipo di trattamento
TAB-ART rappresenta l'insieme dei dati come risulta strutturato in memoria, questo infatti é il modo in cui essi intervengono nel processo di elaborazione
indica che l'unitá "GRUPPO RECORDS MOVIMENTI" di cui si compone l'ARTICOLO può non essere presente
indica il nome del flusso sequenziale da cui é tratta la tabella
ARK-CLI
F.L.R.
CLIENTE-A
COD-CLIEN
ANAGRAFICA
RMOV
FILIALE
CLIENTE
PRESENTE IN ARK-CLI
ASSENTE DA ARK-CLI
TESTATA
GR-REC-MOV
ARTICOLO
MOVIMENTO
PRESENTE
ASSENTE
TRK
COD-FIL
COD-CLIEN
COD-ART
ALTRI - DATI
COD-MOV
QTA
INP
TAB-ART
F.L.R.
(TABE: SEQUENZIALE)
ARTICOLO
COD-ART
PRU
PERC-IVA
C: CLIENTE presente in ARK-CLI
C1: record TRK = 1 presente
C2: record TRK = 2 presente
C3: primo movimento della FILIALE o inizio pagina
C4: primo movimento del CLIENTE o inizio pagina

3. PRIMO PASSO: INDIVIDUAZIONE DELLE STRUTTURE DEI DATI A PARTIRE DALLE SPECIFICHE FUNZIONALI DEL PROBLEMA

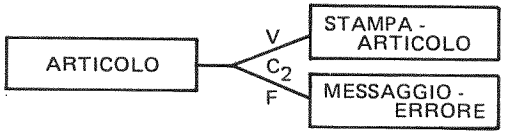
Descrivere le strutture dei dati significa, nell'ambito della metodologia PHOS, individuare al loro interno una organizzazione concettualmente coerente con quella che è la struttura logica del problema, cioè individuare quegli insiemi di dati che sono caratterizzati dall'essere assoggettati ad un trattamento unitario.

I dati vengono descritti sulla base di tre sole strutture organizzative, proprie della programmazione strutturata: sequenza, selezione, iterazione.
La notazione utilizzata è illustrata dai seguanti esempi:

- l'unità logica "CLIENTE-ARCHIVIO" è composta dalla sequenza delle unità logiche "COD-CLIEN" e "ANAGRAFICA":

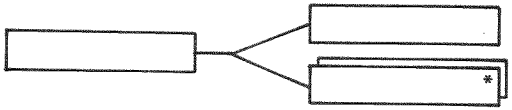


- l'unità logica "ARTICOLO" (in output), può consistere di una segnalazione "MESSAGGIO-ERRORE" o di una "STAMPA-ARTICOLO", a seconda che la condizione C_2 sia vera (V) o falsa (F):

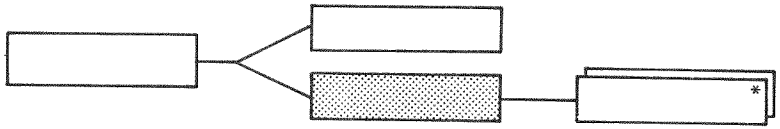


- l'unità logica "FILIALE" è composta da una iterazione di unità logiche "CLIENTE":

La descrizione delle strutture dei dati viene effettuata per livelli di raffinamento successivi, per visualizzarne l'ordinamento gerarchico, utilizzando una sola struttura organizzativa in ogni scomposizione.
Non sono ammesse quindi, per esempio, strutture del tipo:



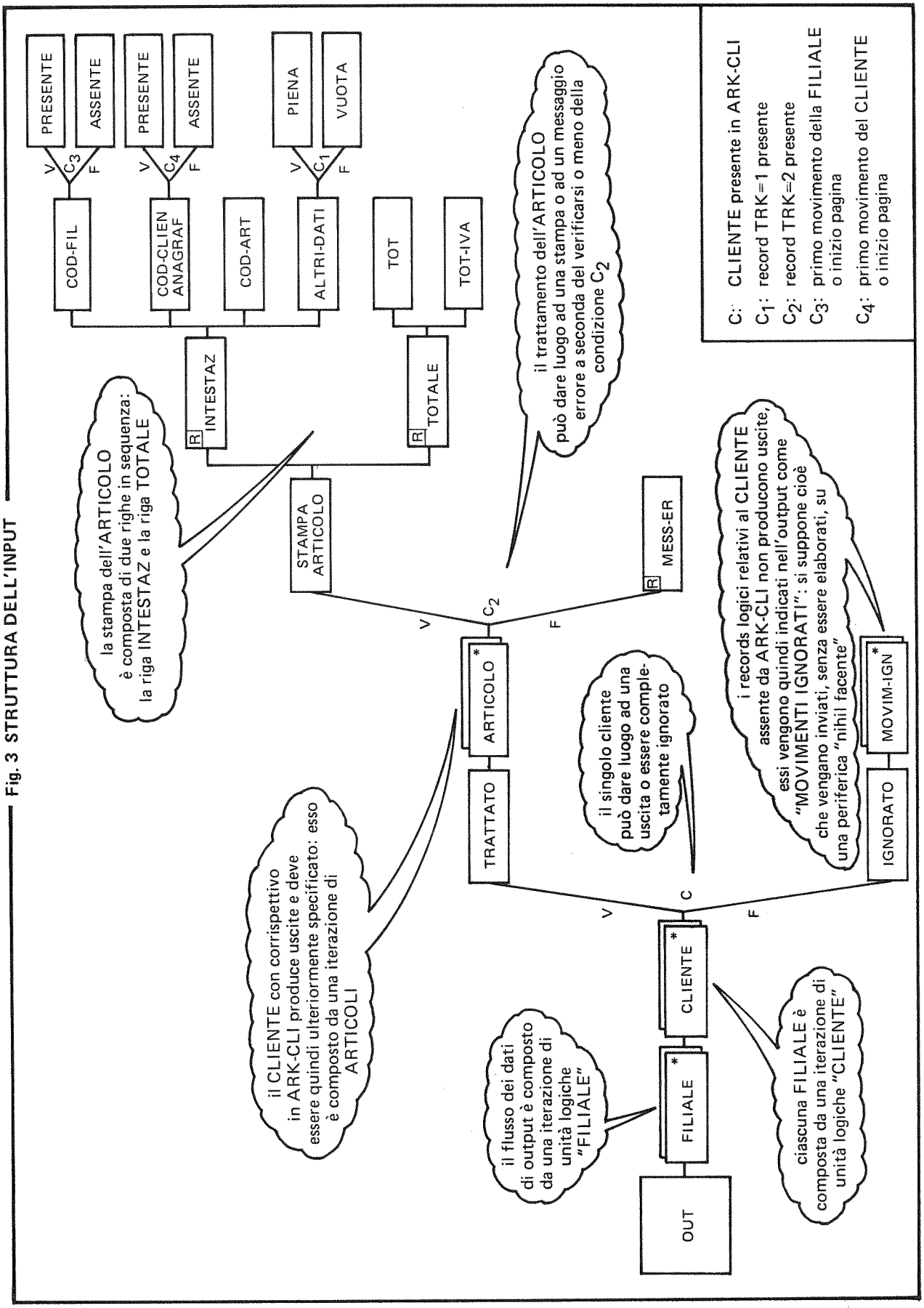
che combinano iterazione e alternativa; tale struttura dovrà invece essere rappresentata con:



INPUT

La scelta che è alla base della metodologia PHOS consiste nel privilegiare, nella descrizione dei dati di input, l'indipendenza dei singoli files: flussi fisici distinti vengono quindi descritti separatamente, mettendone però in evidenza le interrelazioni logiche.
Il problema proposto presenta tre flussi di input: RMOV, TABE, ARK-CLI, che sono descritti secondo le regole grafiche della metodologia nel disegno di fig. 2. E' da notare come la descrizione data dell'unità "CLIENTE" di RMOV metta in evidenza le correlazioni logiche esistenti con le unità corrispondenti del flusso ARK-CLI: queste correlazioni caratterizzano una diversificazione nel trattamento ed è questa la ragione che ne giustifica l'introduzione.

Fig. 3 STRUTTURA DELL'INPUT

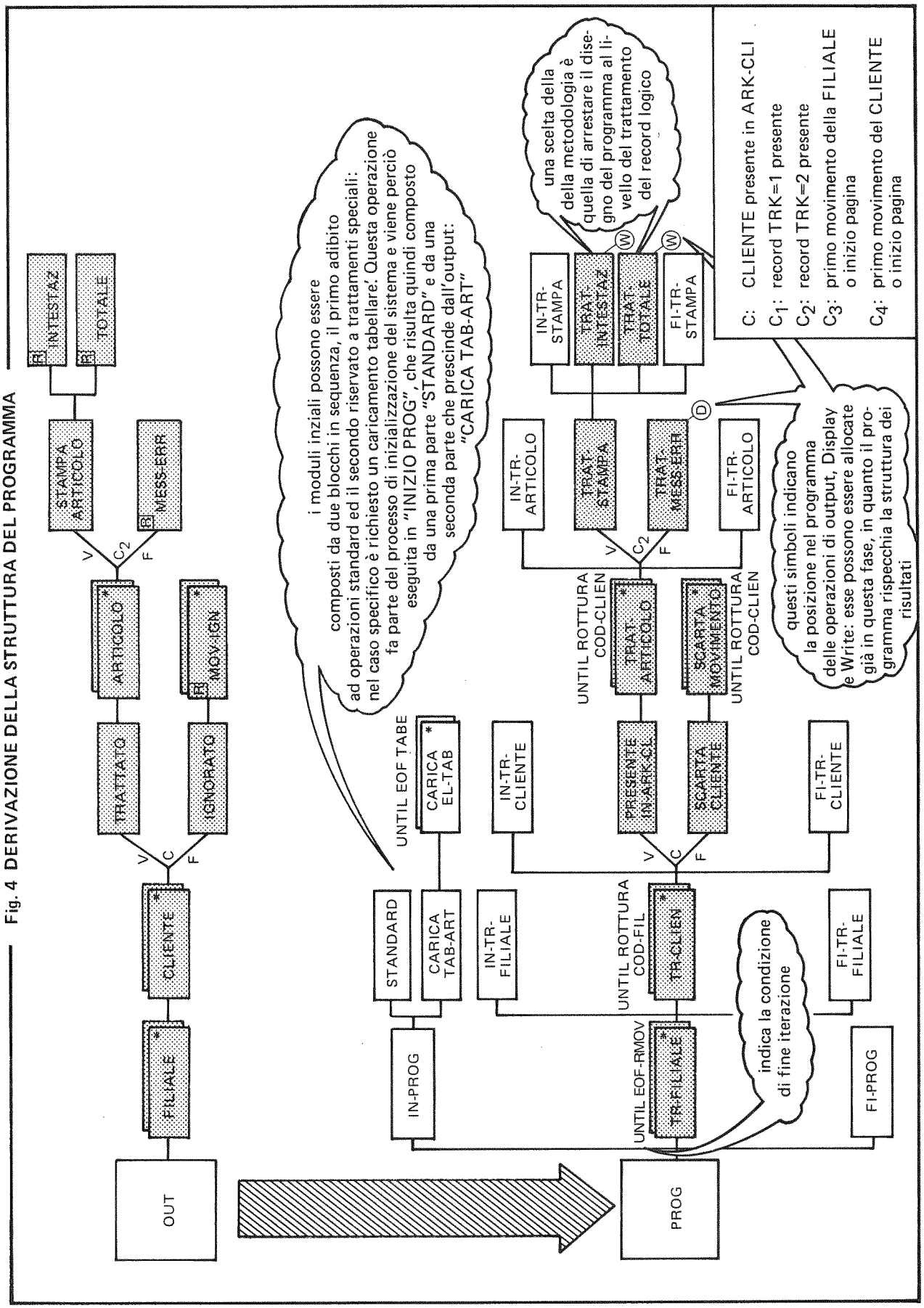


OUTPUT

Sempre alla base della metodologia PHOS sta la scelta di descrivere i dati di output secondo un'unica strttura logica.
Il fatto che in uscita siano presenti flussi fisici distinti è segnalato diversificando, nel disegno dell'output, l'unità logica a seconda del diverso supporto di registrazione; tra le periferiche d'uscita si include anche una periferica "nihil facente" ove vengono idealmente inviate le unità da ignorare.

Nel disegno di fig. 3 è presentata la struttura dei dati di output ottenuta secondo una analisi gerarchica degli insiemi logici di uscita; è da notare la diversificazione introdotta a livello di cliente tra i risultati reali e quelli fittizi, "CLIENTE-TRATTATO" e "CLIENTE-IGNORATO", che precisa le diverse uscite logiche del programma al di là di considerazioni di natura prettamente fisica.

Fig. 4 DERIVAZIONE DELLA STRUTTURA DEL PROGRAMMA



4. SECONDO PASSO: DERIVAZIONE DELLA STRUTTURA DEL PROGRAMMA DA QUELLA DEI DATI DI OUTPUT

Se il primo passo, appena illustrato, è il più importante dal punto di vista dei risultati (la descrizione dei dati condiziona la struttura del programma in modo determinante), il secondo, che sarà oggetto del presente paragrafo, è quello che caratterizza la metodologia PHOS rispetto ad altre metodologie di programmazione.
La scelta di base è di far derivare la struttura portante del programma da quella di output (in effetti questa scelta ha già avuto la sua influenza su quelle fatte nella fase di descrizione dei dati): essa si attua facendo corrispondere ad ogni struttura sequenziale, alternativa, iterativa in output una struttura analoga nel programma, intendendo con ciò una sequenza, selezione, iterazione di operazioni da eseguire piuttosto che di dati.

La fig. 4 mostra, nel caso in esame, le strutture del programma derivate direttamente da quella di output.
Tale struttura resterà sostanzialmente inalterata anche dopo le modifiche che potranno essere introdotte al quarto passo per eliminare le eventuali situazioni conflittuali.

Allo scheletro del programma così derivato si aggiungono dei blocchi di inizio e fine trattamento ad esclusione dei blocchi di programma fittizi introdotti in rispetto alla unicità del criterio di scomposizione, come ad esempio il blocco "PRESENTE IN ARCHIVIO" la cui reale funzione è quella espressa dal modulo successivo "TRATTA ARTICOLO*".
Lo scopo di tali blocchi è di eseguire tutte quelle operazioni che servono per inizializzare o terminare il processo vero e proprio.
Talvolta tale processo di inizializzazione, oltre alle componenti standard (azzeramenti, memorizzazione codici, etc.) deve eseguire dei trattamenti speciali: in tal caso il blocco di "INIZIO TRATTAMENTO" sarà ulteriormente scomposto nella sequenza di un blocco "STANDARD" e di un blocco "TRATTAMENTO SPECIALE"
Nell'esempio specifico, il blocco "IN-PROG" è composto da un blocco "STANDARD" e da un blocco "CARICA-TAB-ART".

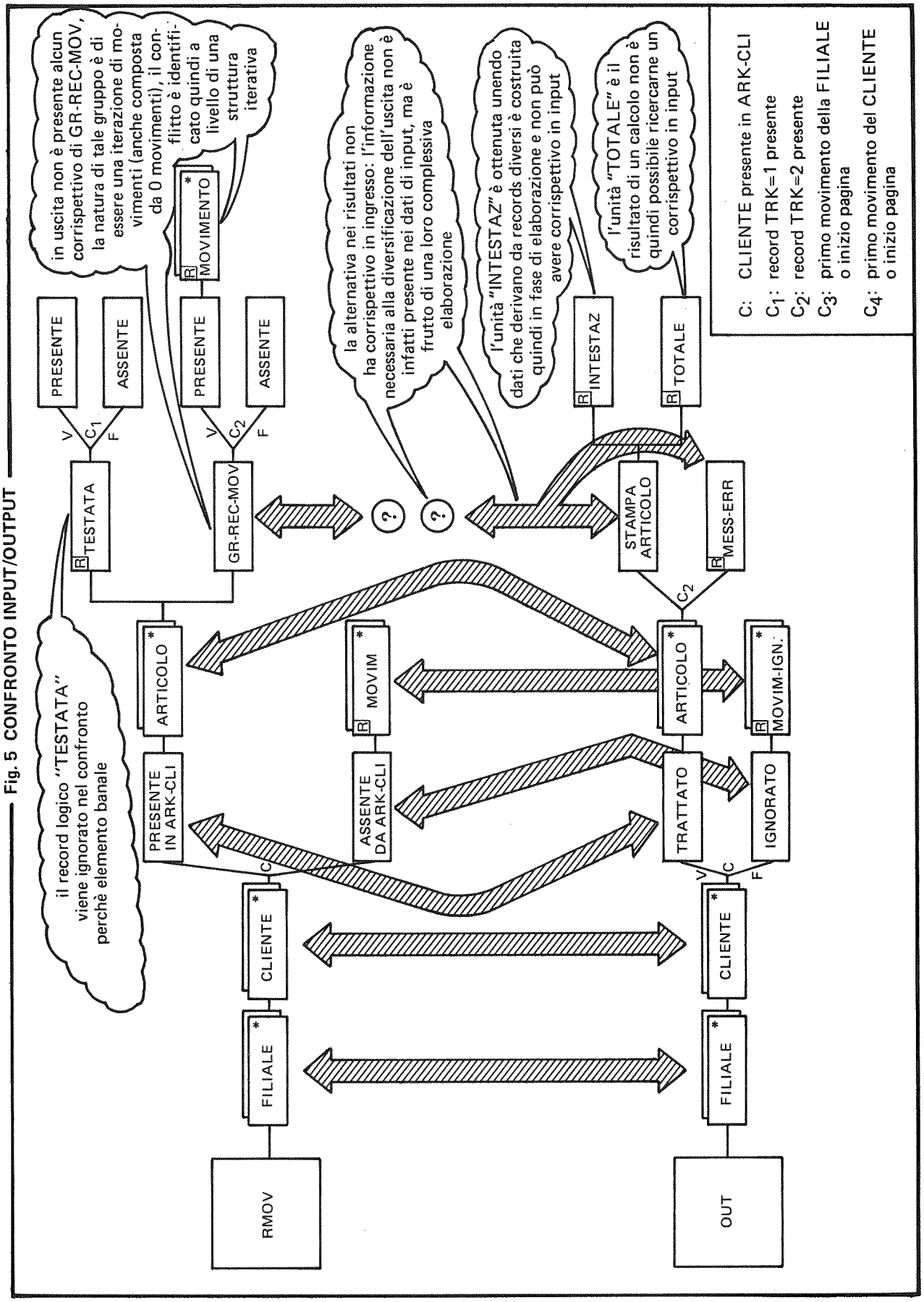


Fig. 5 CONFRONTO INPUT/OUTPUT

5. TERZO PASSO: CONFRONTO DELLE STRUTTURE DI INPUT CON QUELLE DI OUTPUT

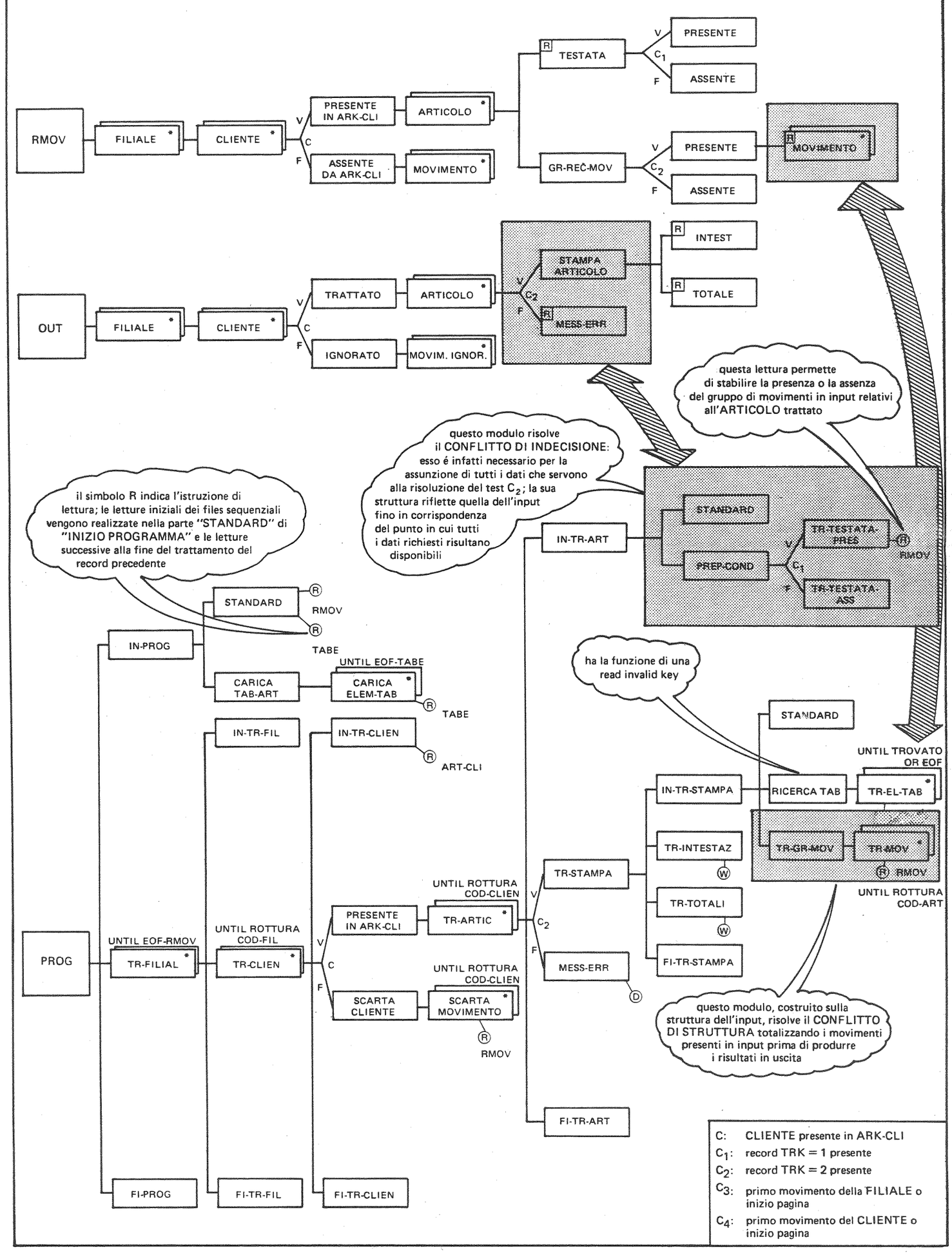
Il programma derivato dal passo precedente risulta completamente indipendente dalle strutture di input e può quindi risultare incompatibile con esse: i dati di input possono cioè presentarsi organizzati in modo diverso da quello che sarebbe necessario per poter essere correttamente trattati dal programma.
Per controllare se questa situazione si verifica e per individuare quali provvedimenti vadano presi per correggerla, si procede all'analisi comparata delle strutture di input con quella di output.
Tale analisi tende ad individuare:

- se tali strutture siano compatibili e se quindi il programma già disegnato possa essere reso operante
- qualora ciò non risulti vero, ove le due strutture sono in conflitto.

Per effettuare tale confronto, PHOS fornisce delle regole precise che possono essere così sintetizzate:

- il confronto va eseguito tra i soli flussi sequenziali di input e il flusso di output, ignorando cioè i flussi trattati in accesso random e le tabelle (tali dati sono sempre disponibili e non possono quindi dar luogo a conflitti);
- nel confronto vanno ignorati sia i blocchi introdotti in rispetto alla unicità del criterio di scomposizione sia quei blocchi, quali "INTESTAZIONE" o "TOTALI", che sono da ritenersi poco significativi;
- nel caso in cui in input sia presente un solo flusso sequenziale, il confronto deve mettere in luce se ad ogni unità logica nella struttura di output corrisponde una ed una sola unità logica nella struttura di input e viceversa. In questo caso il programma già disegnato risulterà corretto;
- nel caso in cui in input siano presenti più flussi sequenziali, il confronto deve mostrare se ad ogni unità logica di ciascuno dei flussi sequenziali di input corrisponde una ed una sola unità logica in output e se, viceversa, a ciascuna unità logica di output corrisponde una unità logica in almeno un flusso sequenziale di input.
  Anche in questo caso, il programma già disegnato risulterà corretto.

Fig. 6 MODIFICA DEL PROGRAMMA



6. QUARTO PASSO: MODIFICA DEL PROGRAMMA PER LA SOLUZIONE DEI CASI DI CONFLITTO

Se, nel corso del confronto di cui sopra, si è identificata una situazione conflittuale, si procede alla conseguente modifica del programma.
Per questo è necessario riconoscere il tipo di conflitto in questione, poichè la soluzione suggerita da PHOS varia in dipendenza della sua natura.
Le modifiche alla struttura del programma sono introdotte senza alterare il corpo centrale del programma stesso, senza modificare cioè quella prima struttura che era stata direttamente dedotta dalle strutture dei dati di output: le variazioni vengono introdotte solo inserendo nei blocchi di "INIZIO TRATTAMENTO" i nuovi moduli per la risoluzione della situazione di conflitto.

In una prima suddivisione si possono riconoscere due categorie di situazioni conflittuali:

- **i conflitti di indecisione**, che si verificano quando la corrispondenza tra input e output viene a cadere a livello di una alternativa;
- **i conflitti di struttura**, che si verificano quando la corrispondenza viene a cadere a livello di una struttura iterativa.

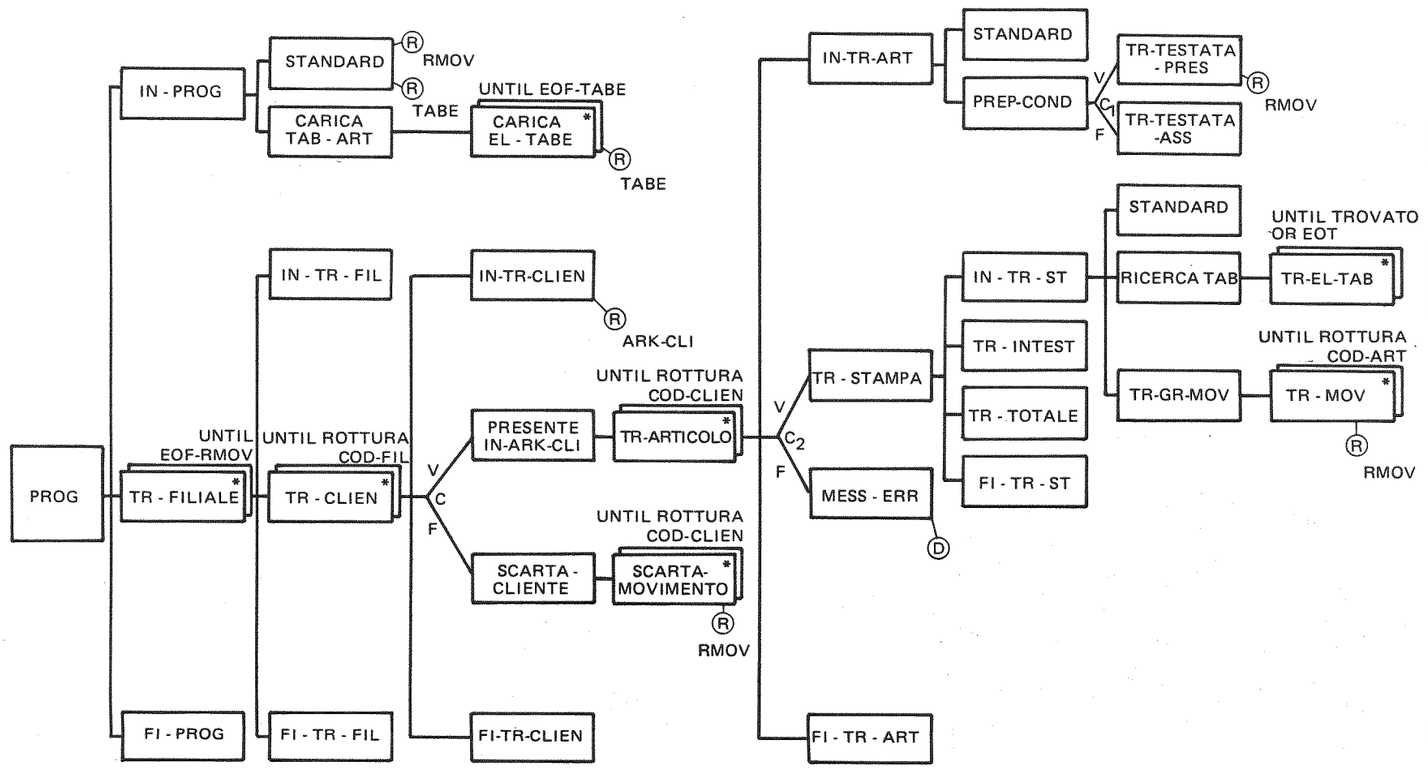
Esempi del verificarsi di tali tipi di conflitto, nell'esempio illustrato (vedi fig. 5), sono la scomposizione dell'unità "ARTICOLO" di output (che dà luogo ad una alternativa assente in input) e la presenza in input di una iterazione di records movimento che non ha corrispettivo in uscita.

La soluzione proposta per il primo tipo di situazioni è di introdurre, nel blocco di "INIZIO TRATTAMENTO" dell'unità la cui scomposizione dà luogo al conflitto, un blocco che svolga la funzione di preparare le informazioni necessarie alla risoluzione della alternativa.
I conflitti di indecisione infatti derivano dal fatto che, nel punto in cui si presenta l'alternativa assente in input, manca l'informazione necessaria a rispondere al test che la pilota.
Il blocco di risoluzione del conflitto, chiamato con il nome indicativo di "PREPARA CONDIZIONE", ha il compito di assumere dall'input tutte le informazioni necessarie per la risoluzione del test. Deve quindi rispecchiare quella parte della struttura dell'input che descrive appunto i dati da assumere, questo poichè il modo in cui essi si presentano, la loro organizzazione appunto, è condizionante ai fini dell'elaborazione.

Più articolata si presenta la soluzione dei conflitti di struttura, in quanto essa si differenzia a seconda che l'iterazione sia assente in output oppure in input.

- Nel primo caso, vedi fig. 5, è presente quindi un insieme di dati il cui trattamento non è previsto nella struttura del programma dedotta direttamente da quella di output.
  Tale insieme di dati deve essere quindi elaborato a parte, da un modulo che verrà inserito nel blocco di "INIZIO TRATTAMENTO" del livello precedente all'unità conflittuale. Questo modulo risente del modo in cui i dati di input, in esso trattati, si presentano, dovrà quindi rispecchiare quella parte della struttura dell'input che li descrive.
- Nel caso in cui l'iterazione conflittuale compaia in output, vi è una ulteriore articolazione delle soluzioni proposte nel cui merito non si entrerà in questa sede. Il trattamento risultante sarà comunque realizzato, mantenendo inalterato il corpo centrale del programma dedotto dall'output, nel solito modulo di "INIZIO TRATTAMENTO" del livello corrispondente all'unità conflittuale; la sua strutturazione sarà differente a seconda dello specifico caso in esame e potrà richiedere l'ulilizzo di un file intermedio per la ristrutturazione dell'input.

Fig. 7 ALLOCAZIONE FUNZIONI ELEMENTARI



FI-PROG

CLOSE files

STANDARD-IN-PROG

OPEN files
ﬀ ⟶ area-st
1 ⟶ CONTA-RIGHE
0 ⟶ NUM-ELEM-TAB
READ RMOV
READ TABE

IN-TR-FIL

COD-FIL ⟶ DEP-FIL
"PRIMO" ⟶ IND-PR-FIL

FI-TR-FIL

no operation

CARICA-EL-TABE

NUM-ELEM-TAB = NUM-ELEM-TAB + 1
record TABE ⟶ ELEM-TABE (NUM-ELEM-TAB)
READ TABE

IN-TR-CLIEN

COD-CLIEN ⟶ DEP-CLIEN
"PRIMO" ⟶ IND-PR-CLI
READ ARK-CLI

FI-TR-CLIEN

no operation

SCARTA-MOV

READ RMOV

MESS-ERR

DISPLAY COD-ART

FI-TR-ART

no operation

STANDARD-ART

COD-ART ⟶ DEP-ART
0 ⟶ TOTALE

TR-INTEST

COD-ART ⟶ area-st
TESTATA ⟶ area-st
IF CONTA-RIGHE > 50 OR
  IND-PR-FIL = "PRIMO"
  DEP-FIL ⟶ area-st
  DEP-CLIEN ⟶ area-st
  ANAGRAFIC ⟶ area-st
  "ALTRO" ⟶ IND-PR-FIL
  "ALTRO" ⟶ IND-PR-CLI
ELSE
  IF IND-PR-CLI = "PRIMO"
    DEP-CLIEN ⟶ area-st
    ANAGRAFIC ⟶ area-st
    "ALTRO" ⟶ IND-PR-CLI
  ELSE
    no operation
IF CONTA-RIGHE > 50
  salto pagina
  0 ⟶ CONTA-RIGHE
ELSE no operation
WRITE RIGA-INTEST
ﬀ ⟶ RIGA-INTEST
CONTA-RIGHE = CONTA-RIGHE + 1

TR-TOTALI

TOTALE ⟶ area-st
IVA = TOTALE * PERC (IND-ELEM-TAB) / 100
WRITE RIGA-TOTALI
ﬀ ⟶ RIGA-TOTALI
CONTA-RIGHE = CONTA-RIGHE + 1

FI-TR-ST

no operation

TR-TESTATA-PRES

TESTATA ⟶ area-st
READ RMOV

TR-TESTATA-ASS

no operation

STANDARD-ST

0 ⟶ IND-ELEM-TAB
"NO" ⟶ TROVATO

TR-ELEM-TAB

IND-ELEM-TAB = IND-ELEM-TAB + 1
IF COD-ART = COD-ART (IND-ELEM-TAB)
  "SI" ⟶ TROVATO

TR-MOV

IMPONIBILE = QTA * PRU (IND-ELEM-TAB)
TOTALE = IMPONIBILE + TOTALE
READ RMOV

la allocazione delle funzioni elementari viene realizzata livello per livello, assegnando le varie istruzioni ai soli blocchi terminali; tali istruzioni coincidono con quelle standard di un linguaggio evoluto

C: CLIENTE presente in ARK - CLI
C_1: record TRK = 1 presente
C_2: record TRK = 2 presente
C_3: primo movimento della FILIALE o inizio pagina
C_4: primo movimento del CLIENTE o inizio pagina

7. QUINTO PASSO: ALLOCAZIONE DELLE FUNZIONI ELEMENTARI

Il risultato della esecuzione dei passi precedenti è la individuazione della struttura completa del programma, cioè il disegno della sua organizzazione logica.
Il passo successivo è quello di inserire nei singoli blocchi le varie istruzioni: specificare cioè, in termini di operazioni elementari, la funzione che ciascuno di essi deve eseguire.
L'insieme di istruzioni utilizzate è compatibile, per assicurare l'indipendenza di questa fase dal linguaggio di codifica, con i linguaggi di programmazione più in uso ed è espresso, per quanto è possibile, in linguaggio corrente.

Tali istruzioni vanno allocate nei blocchi terminali, nei blocchi cioè che non vengono ulteriormente sviluppati.
In particolare, nei blocchi iniziali e finali vanno allocate le funzioni elementari che inizializzano o terminano il processo, cioè:

. nel blocco di inizio programma:
  - apertura flussi
  - abblencamenti ed azzeramenti iniziali
  - lettura del primo record dei flussi sequenziali

. nel blocco di fine programma:
  - chiusura flussi

. nel blocco di inizio trattamento intermedio:
  - memorizzazione codici
  - azzeramenti locali
  - lettura dei flussi random

. nel blocco di fine trattamento intermedio:
  - scritture e conseguenti abblencamenti delle aree di scrittura
  - letture dei records successivi dei flussi sequenziali.

Tale processo viene eseguito in modo Top-Down, assegnando ai blocchi terminali di ciascun livello le istruzioni che gli competono (vedi fig. 7).

8. SESTO PASSO: CODIFICA STANDARD IN COBOL

Benchè il metodo sviluppato sia indipendente dal linguaggio di programmazione, i problemi alla cui soluzione esso è rivolto sono quelli tipici dell'ambiente EDP, abitualmente codificati in Cobol: la metodologia propone quindi uno standard di codifica in questo linguaggio, le cui linee guida sono:

a) Il disegno del programma viene codificato secondo un procedimento di tipo Top-Down in un'unica SECTION composta di più parti, ciascuna corrispondente ad un singolo livello gerarchico.
Tale SECTION, residente in memoria, cui si assegna il nome convenzionale di MAIN SECTION, realizza il controllo dell'intero programma.
Il processo di stesura di tale sezione viene effettuato secondo i seguenti criteri:

   . ai moduli costituenti una struttura sequenziale si associa una sequenza di istruzioni "PERFORM...";

   . ai moduli costituenti una struttura alternativa si associa una istruzione "IF...PERFORM...ELSE PERFORM..."

   . ai moduli costituenti una struttura iterativa si associa l'istruzione "PERFORM... UNTIL...".

b) Le istruzioni elementari, che possono comporre solo blocchi terminali, vengono codificate in SECTION separate: ad ogni blocco corrisponde una sezione.

c) I blocchi in cui non deve essere eseguita alcuna operazione vengono codificati come commenti (vedi fig. 8).

d) I blocchi rappresentati nel programma solo in rispetto alla regola della unicità della scomposizione vengono aboliti in fase di codifica.

Riferimenti:

(1) T. Bettinazzi et al. : "PHOS: una metodologia per la costruzione veloce ed affidabile di programmi"
Note di Software n. 4

(2) E. Toscani : "PHOS: una metodologia per la costruzione di programmi in ambiente E.D.P."
Tesi di Laurea, Università di Milano 1978

Fig. 8 IDENTIFICATION-ENVIRONMENT DIVISIONS

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  RENDIC.
*
*****************************************************************
*                                                               *
*      QUESTO PROGRAMMA FORNISCE UN RENDICONTO DEI MOVIMENTI    *
*      RELATIVI AD UN ARTICOLO, FORNENDO I TOTALI DI OGNI ARTICOLO*
*      QUESTO PROGRAMMA FORNISCE UN RENDICONTO DEI MOVIMENTI    *
*      ALL' INTERNO DI CLIENTE E FILIALE                        *
*                                                               *
*****************************************************************
*
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  LEVEL-62 WITH DEBUGGING MODE.
 OBJECT-COMPUTER.  LEVEL-62.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT TABE ASSIGN TO TABE-MSU310.
     SELECT ARK-CLI ASSIGN TO ARK-MSU310
                    ORGANIZATION IS INDEXED
                    ACCESS MODE IS RANDOM
                    RECORD KEY IS   COD-CLIEN.
     SELECT RMOV ASSIGN TO RMOV-MSU310.
     SELECT STAMPA  ASSIGN TO STAMPA-REPORT.
```

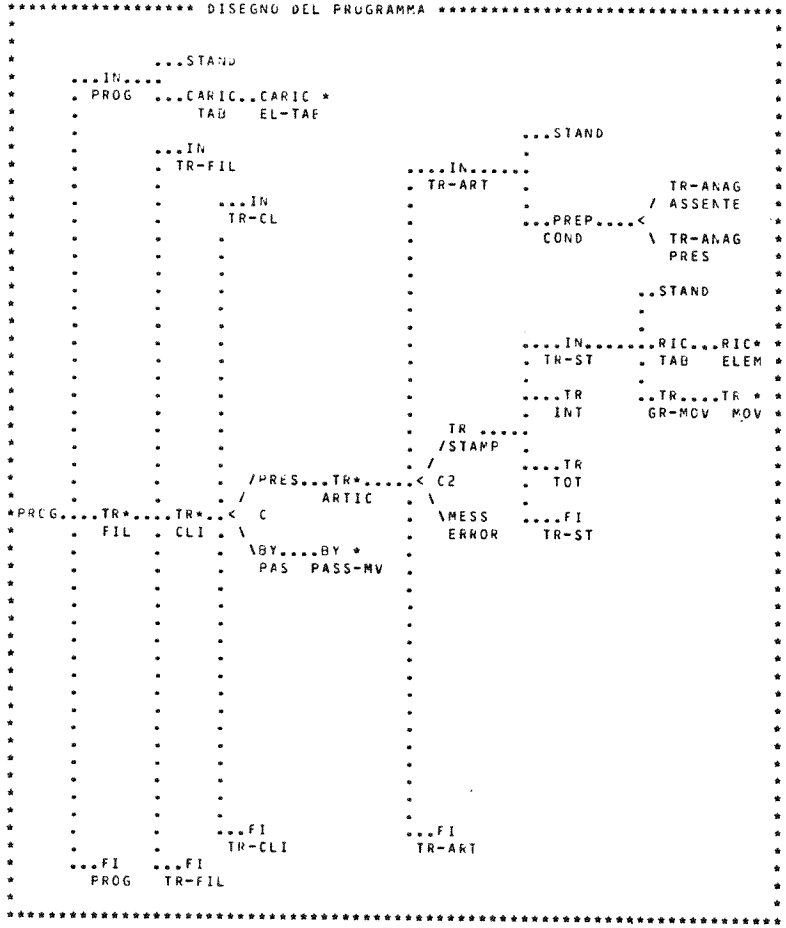


Fig. 9 DATA DIVISION - FILE SECTION

```
/
 DATA DIVISION.
 FILE SECTION.
*
*****************************************************************
*                                                               *
*               "DESCRIZIONE DATI DI INPUT"                     *
*                                                               *
*****************************************************************
*
***** TABE- FLUSSO SEQUENZIALE CONTENENTE I RECORDS CON DATI     *
*****       RELATIVI AL SINGOLO ARTICOLO  (TIPO-RECORD, CODICE- *
*****       ARTICOLO, PREZZO UNITARIO, PERCENTUALE-IVA) DA      *
*****       CARICARE IN TABELLA.                                *
*
 FD  TABE     LABEL RECORD IS STANDARD
              DATA RECORD IS ELEMENTO-TABELLA.
 01  ELEMENTO-TABELLA.
     02  TRK-T              PIC X.
     02  COD-ART            PIC X(8).
     02  PRZ                PIC 9(6).
     02  IVA                PIC 99.
*
*
*
***** ARK-CLI- FLUSSO INDEXED CONTENENTE RECORDS CON DATI       *
*****          RELATIVI AI SINGOLI CLIENTE DA UTILIZZARSI IN    *
*****          ACCESSO RANDOM PER LA RICERCA DEI DATI ANAGRAFICI *
 FD  ARK-CLI LABEL RECORD IS STANDARD
             DATA RECORD IS REC-ARK.
 01  REC-ARK.
     02  TRK-ARK            PIC X.
     02  COD-CLIEN          PIC X(8).
     02  ANAGRAFICA         PIC X(30).
*
*
*
***** RMOV- FLUSSO SEQUENZIALE ORDINATO SU FILIALE,CLIENTE E     *
*****       ARTICOLO. OGNI ARTICOLO E COMPOSTO DA UN RECORD     *
*****       DESCRIZIONE E DA UN GRUPPO DI RECORDS MOVIMENTO     *
*****       SIA LA DESCRIZIONE CHE IL GRUPPO MOVIMENTI PUO ESSERE*
*****       EVENTUALMENTE ASSENTE                               *
*
 FD  RMOV     LABEL RECORD IS STANDARD
              DATA RECORD IS REC-MOV.
 01  REC-MOV.
     02  TRK                PIC 9.
     02  COD-ART-MOV.
         03  COD-CLIEN-MOV.
             04  COD-FIL-MOV PIC X(8).
             04  KEY-CLIEN   PIC X(8).
         03  CODICE-ART      PIC X(6).
     02  DESCR              PIC X(30).
     02  MOVIMENTO REDEFINES DESCR.
         03  COD-MOV         PIC X(8).
         03  QTA             PIC 9(4).
         03  FILLER          PIC X(18).
*
*****************************************************************
*                                                               *
*               "DESCRIZIONE DATI DI OUTPUT"                    *
*                                                               *
*****************************************************************
*                                                               *
*
***** STAMPA-FLUSSO SU LINE PRINTER CONTENENTE IL RENDICONTO DEI *
***** MOVIMENTI RELATIVI A CIASCUN ARTICOLO ORDINATO SU FILIALE  *
***** E CLIENTE: PER CIASCUN ARTICOLO SONO PRESENTI DUE RIGHE DI *
***** STAMPA,UNA RIGA INTESTAZIONE ED UNA RIGA CON LA TOTALIZZA- *
***** ZIONE DEI MOVIMENTI                                       *
*
*
 FD  STAMPA  LABEL RECORD IS OMITTED
             DATA  RECORD ARE RIGA-INTEST  RIGA-TOT.
 01  RIGA-INTEST.
     02  COD-FIL-ST          PIC BBX(8).
     02  COD-CLIEN-ST        PIC BBX(8).
     02  ANAG-CLIEN-ST       PIC BBX(30).
     02  COD-ART-ST          PIC BBX(8).
     02  DESC-ART-ST         PIC BBX(30).
*
 01  RIGA-TOT.
     02  TOT-ART-ST          PIC B(62)9(11).
     02  IVA-ST              PIC B(4)9(11).
     02  FILLER              PIC X(6).
```

Fig. 10 DATA DIVISION-WORKING-STORAGE SECTION

```
*
 WORKING-STORAGE SECTION.
*
*****************************************************************
*                                                               *
*     DESCRIZIONE CAMPI DI LAVORO: CAMPI DI FINE FLUSSI,CAMPI   *
*     CONDIZIONE,CAMPI DEPOSITO CODICE,TABELLA ARTICOLI E SUOI  *
*     CAMPI AUSILIARI MESSAGGIO ERRORE                          *
*                                                               *
*****************************************************************
*
*
 01  IND-EOF-MOV              PIC X(7) VALUE IS "NONTROV".
     88 EOF-MOV                        VALUE IS "TROVATO".
*
 01  IND-EOF-TABE             PIC X(7) VALUE IS "NONTROV".
     88 EOF-TABE                       VALUE IS "TROVATO".
*
 01  DEP-COD-ART.
     02  DEP-CLIEN.
         03  DEP-COD-FIL      PIC X(8).
         03  DEP-KEY-CLIEN    PIC X(8).
     02  DEP-CODICE-ART       PIC X(8).
*
 01  TABELLA-ARTICOLI.
     02  EL-TABE              OCCURS 50 TIMES.
      03 TRK-TAB             PIC X.
         03  COD-ART-TAB      PIC X(8).
         03  PRU-TAB          PIC 9(6).
         03  PERC-IVA-TAB     PIC 99.
*
 01  IND-EL-TAB               PIC 99.
*
 01  NUM-EL-TAB               PIC 99.
*
 01  MESSAGGIO-ERR            PIC X(32)  VALUE IS "ASSENZA MOVIMENT
-                                 "I PER COD-ART=  ".
*
 01  TROVATO-IN-ARK           PIC XX.
     88  C                    VALUE IS "SI".
*
 01  GR-MOV                   PIC X(8).
     88  C2                   VALUE IS "PRESENTE".
*
*****************************************************************
*                                                               *
*     DESCRIZIONE DEI CAMPI LAVORO :                            *
*     DEPOSITO DESCRIZIONE ARTICOLO,CONTATORE RIGHE DI STAMPA,  *
*     TOTALE IMPONIBILE ARTICOLO,IMPONIBILE ARTICOLO (PRU*QTA), *
*     INDICATORE DEL PRIMO CLIENTE DELLA FILIALE,INDICATORE DEL *
*     PRIMO ARTICOLO DEL CLIENTE,INDICATORE DI ARTICOLO TROVATO *
*     IN TABELLA                                                *
*                                                               *
*****************************************************************
*
 01  DEP-DESCR                PIC X(30).
*
 01  CONTA-RIGHE              PIC 99.
*
 01  TOTALE                   PIC 9(11).
*
 01  IMPONIBILE               PIC 9(10).
*
 01  INDICE-CLIENTE           PIC X(16).
*
 01  INDICE-ARTICOLO          PIC X(22).
*
 01  TROVATO-IN-TAB           PIC XX.
```

Fig. 11 PROCEDURE DIVISION: CODIFICA DELL'ALBERO DEL PROGRAMMA

```
PROCEDURE DIVISION.
MAIN SECTION
ALBERO-DEL-PROGRAMMA.
*
*****************************************************************
*                                                               *
*     INPUT :FLUSSO TABE UTILIZZATO PER LA COSTRUZIONIONE DI UNA *
*            TABELLA                                            *
*            FLUSSO RMOV ORDINATO PER FILIALE,CLIENTE E ARTICOLO *
*            CONTENENTE PER OGNI ARTICOLO UN RECORD INTESTAZIONE *
*            E UNA SERIE DI RECORDS MOVIMENTO                   *
*            FLUSSO ARK-CLI,RANDOM SULLA CHIAVE COD-CLIEN,CONTE- *
*            NENTE UN RECORD PER OGNI CLIENTE                   *
*     OUTPUT:TABULATO TOTALIZZAZIONI ARTICOLO                   *
*     ELABOR:SCORRERE SEQUENZIALMENTE IL FLUSSO RMOV TOTALIZZANDO*
*            LE QUANTITA (QTA) PER OGNI ARTICOLO.               *
*            I CLIENTI SENZA CORRISPETTIVO IN ARK-CLI VANNO IGNO-*
*            RATI-GLI ARTICOLI SENZA MOVIMENTI (CIOE'CON IL SOLO *
*            RECORD DESCRIZIONE) NON DEVONO ESSERE STAMPATI E UN *
*            MESSAGGIO DI ERRORE DEVE ESSERE INVIATO SU CONSOLLE *
*                                                               *
*****************************************************************
*
*
     PERFORM IN-PROG
     PERFORM TR-FIL UNTIL EOF-MOV
     PERFORM FI-PROG
     STOP RUN.
*
*****************************************************************
*                                                               *
*     SECONDO LIVELLO DEL DISEGNO PROGRAMMA                     *
*                                                               *
*****************************************************************
*
*
 IN-PROG.
     PERFORM STANDARD-IN-PROG
     PERFORM CARICA-ELEM UNTIL EOF-TABE.
*
 TR-FIL.
     PERFORM IN-TR-FIL
     PERFORM TR-CLIEN UNTIL COD-FIL-MOV NOT EQUAL DEP-COD-FIL
                         OR EOF-MOV.
*    PERFORM FI-TR-FIL.
*
*****************************************************************
*                                                               *
*     TERZO LIVELLO DEL DISEGNO PROGRAMMA                       *
*                                                               *
*****************************************************************
*
*
 TR-CLIEN.
     PERFORM IN-TR-CLIEN.
     IF C PERFORM TR-ART UNTIL COD-CLIEN-MOV NOT EQUAL DEP-CLIEN
                            OR EOF-MOV
     ELSE PERFORM BY-PASS UNTIL COD-CLIEN-MOV NOT EQUAL DEP-CLIEN
                            OR EOF-MOV.
*    PERFORM FI-TR-CLIEN.
*
*****************************************************************
*                                                               *
*     QUARTO LIVELLO DISEGNO PROGRAMMA                          *
*                                                               *
*****************************************************************
*
*
 TR-ART.
     PERFORM IN-TR-ART
     IF C2 PERFORM TR-STAMPA
     ELSE DISPLAY MESSAGGIO-ERR DEP-COD-ART.
*    PERFORM FI-TR-ART.
*
*****************************************************************
*                                                               *
*     QUINTO LIVELLO DISEGNO PROGRAMMA                          *
*                                                               *
*****************************************************************
*
*
 TR-STAMPA.
     PERFORM IN-TR-STAMPA
     PERFORM TR-INTEST
     PERFORM TR-TOTALI  .
*    PERFORM FI-TR-STAMPA.
*
*
*****************************************************************
*                                                               *
*     SESTO LIVELLO DEL DISEGNO PROGRAMMA                       *
*                                                               *
*****************************************************************
*
*
 IN-TR-STAMPA.
     PERFORM STANDARD-IN-TR-STAMPA
     PERFORM TR-EL-TAB UNTIL TROVATO-IN-TAB = "SI" OR
                             NUM-EL-TAB = IND-EL-TAB
     PERFORM TR-MOV UNTIL COD-ART-MOV NOT EQUAL DEP-COD-ART
                         OR EOF-MOV.
```

Fig. 12 PROCEDURE DIVISION: FUNZIONI ELEMENTARI

```
*
*****************************************************************
*                                                               *
*     ALLOCAZIONE FUNZIONI ELEMENTARI: PRIMO LIVELLO            *
*                                                               *
*****************************************************************
*
*
 FI-PROG SECTION.
 FINE-PROGRAMMA.
     CLOSE TABE  RMOV ARK-CLI STAMPA.
*
*****************************************************************
*                                                               *
*     ALLOCAZIONE FUNZIONI ELEMENTARI : SECONDO LIVELLO         *
*                                                               *
*****************************************************************
*
*
 STANDARD-IN-PROG SECTION .
 STANDARD-INIZIO-PROG.
     OPEN INPUT TABE  RMOV ARK-CLI
     OPEN OUTPUT STAMPA
     MOVE SPACES TO RIGA-INTEST
     MOVE ZERO TO NUM-EL-TAB
     MOVE 1 TO CONTA-RIGHE
     READ RMOV AT END MOVE "TROVATO" TO IND-EOF-MOV.
     READ TABE AT END MOVE "TROVATO" TO IND-EOF-TABE.
*
 CARICA-ELEM SECTION.
 CARICA-ELEM-TABELLA.
     ADD 1 TO NUM-EL-TAB
     MOVE ELEMENTO-TABELLA TO EL-TABE (NUM-EL-TAB)
     READ TABE AT END MOVE "TROVATO" TO IND-EOF-TABE.
*
 IN-TR-FIL SECTION.
 INIZIO-TR-FILIALE.
     MOVE COD-FIL-MOV TO DEP-COD-FIL
     MOVE "PRIMO-CL-FILIALE" TO INDICE-CLIENTE
*
*FI-TR-FIL SECTION.
*FINE-TR-FILIALE.
*    NO OPERATION.
*
*
*****************************************************************
*                                                               *
*     ALLOCAZIONE FUNZIONI ELEMENTARI: TERZO LIVELLO            *
*                                                               *
*****************************************************************
*
*
 IN-TR-CLIEN SECTION.
 INIZIO-TR-CLIENTE.
     MOVE KEY-CLIEN TO DEP-KEY-CLIEN
     MOVE KEY-CLIEN TO COD-CLIEN
     MOVE "SI" TO TROVATO-IN-ARK
     READ ARK-CLI INVALID KEY MOVE  "NO" TO TROVATO-IN-ARK.
     MOVE "PRIMO-ARTICOLO-CLIENTE" TO INDICE-ARTICOLO
*
 BY-PASS SECTION.
 BY-PASS-CLIENTE.
     READ RMOV AT END MOVE "TROVATO" TO IND-EOF-MOV.
*
*FI-TR-CLIEN SECTION.
*FINE-TR-CLIENTE.
*    NO OPERATION.
*
*****************************************************************
*                                                               *
*     ALLOCAZIONE FUNZIONI ELEMENTARI : QUARTO LIVELLO          *
*                                                               *
*****************************************************************
*
*
 IN-TR-ART SECTION.
 INIZIO-TR-ARTICOLO.
     MOVE CODICE-ART TO DEP-CODICE-ART
     MOVE SPACES TO DEP-DESCR.
     IF TRK = 1 MOVE DESCR TO DEP-DESCR
                READ RMOV AT END MOVE "TROVATO" TO IND-EOF-MOV.
*    ELSE NEXT SENTENCE.
     IF TRK = 2 AND COD-ART-MOV = DEP-COD-ART
     MOVE "PRESENTE" TO GR-MOV
     ELSE  MOVE "ASSENTE "      TO GR-MOV.
*
*FI-TR-ART SECTION.
*FINE-TR-ARTICOLO.
*    NO OPERATION.
```

Fig. 13 PROCEDURE DIVISION: FUNZIONI ELEMENTARI (cont.)

```
*
*****************************************************************
*                                                               *
*     ALLOCAZIONE FUNZIONI ELEMENTARI : QUINTO LIVELLO          *
*                                                               *
*****************************************************************
*
 TR-INTEST SECTION.
 TRATTAMENTO-INTESTAZ.
     MOVE DEP-CODICE-ART TO COD-ART-ST
     MOVE DEP-DESCR TO DESC-ART-ST
     IF CONTA-RIGHE > 50 OR INDICE-CLIENTE = "PRIMO-CL-FILIALE"
        MOVE DEP-COD-FIL TO COD-FIL-ST
        MOVE DEP-KEY-CLIEN TO COD-CLIEN-ST
        MOVE ANAGRAFICA  TO ANAG-CLIEN-ST
        MOVE "ALTRO-CL-FILIALE" TO INDICE-CLIENTE
        MOVE "ALTRO-ARTICOLO-CLIENTE" TO INDICE-ARTICOLO
     ELSE
         IF INDICE-ARTICOLO = "PRIMO-ARTICOLO-CLIENTE"
             MOVE DEP-KEY-CLIEN TO COD-CLIEN-ST
             MOVE ANAGRAFICA TO ANAG-CLIEN-ST
             MOVE "ALTRO-ARTICOLO-CLIENTE" TO INDICE-ARTICOLO.
     IF CONTA-RIGHE NOT > 50 ADD 1 TO CONTA-RIGHE
                             WRITE RIGA-INTEST .
     IF CONTA-RIGHE > 50     MOVE 1 TO CONTA-RIGHE
                             WRITE RIGA-INTEST AFTER ADVANCING
                             PAGE .
     MOVE SPACES TO RIGA-INTEST.
*
 TR-TOTALI SECTION.
 TRATTAMENTO-TOTALI.
     MOVE TOTALE TO TOT-ART-ST
     COMPUTE IVA-ST = (TOTALE * PERC-IVA-TAB (IND-EL-TAB) ) / 100
     WRITE RIGA-TOT
     MOVE SPACES TO RIGA-TOT
     ADD 1 TO CONTA-RIGHE.
*
*FI-TR-STAMPA SECTION.
*FINE-TR-STAMPA.
*    NO OPERATION.
*
*****************************************************************
*                                                               *
*     ALLOCAZIONE FUNZIONI ELEMENTARI : SESTO LIVELLO           *
*                                                               *
*****************************************************************
*
*
 STANDARD-IN-TR-STAMPA SECTION.
 STAND-INIZIO-TR-STAMPA.
     MOVE ZERO TO IND-EL-TAB
     MOVE ZERO TO TOTALE
     MOVE "NO" TO TROVATO-IN-TAB.
*
 TR-EL-TAB SECTION.
 RICERCA-ELEMENTO-TAB.
     ADD 1 TO IND-EL-TAB
     IF DEP-CODICE-ART = COD-ART-TAB (IND-EL-TAB)
        MOVE "SI" TO TROVATO-IN-TAB
     ELSE NEXT SENTENCE.
*
 TR-MOV SECTION.
 TR-MOVIMENTO.
     MULTIPLY PRU-TAB (IND-EL-TAB) BY QTA GIVING IMPONIBILE
     ADD IMPONIBILE TO TOTALE
     READ RMOV AT END MOVE "TROVATO" TO IND-EOF-MOV.
```

IL SIMULA 67

COME LINGUAGGIO DI SPECIFICA ESEGUIBILE

P. Cavallari(°), T. Pedrotti(°), F. Tisato(+)

Riassunto. Il SIMULA 67 presenta caratteristiche tali da renderlo atto all'uso come linguaggio di specifica eseguibile. In particolare, il concetto di classe viene richiamato, e se ne discute l'impiego per strutturare programmi mediante livelli linguistici, contesti linguistici e insiemi di astrazioni, con esempi tratti da un caso reale.

1. INTRODUZIONE

L'argomento discusso in questo lavoro si inserisce nel quadro di un progetto per lo sviluppo di un sistema di specifica e documentazione basato sull'uso del SIMULA 67 (sistema APM/Simula).

Gli obiettivi fondamentali sono:
- sviluppo modulare dei programmi;
- sviluppo top-down del software;
- verifica automatica della congruenza delle interfacce;
- produzione di codice eseguibile, per avere una verifica della congruenza semantica;
- gestione automatica delle librerie di moduli.

E' utile sottolineare l'importanza della generazione, mediante il compilatore Simula, di codice eseguibile su macchina ospite. Ciò consente di introdurre facilities di tracing, invarianti per la verifica di correttezza, trappole per misure prestazionali eccetera; tutte possibilità che, se inserite nel codice di implementazione, comportano la degradazione delle prestazioni, o almeno la necessità di generare copie specializzate.

L'idea di fondo è quella di eseguire su sistema ospite il codice prodotto con il linguaggio di disegno, in modo da scaricare su questo sistema una buona parte dei costi di messa a punto dei programmi, oltre che quelli per il disegno del sistema in senso più astratto. Questo, in concreto, significa anche e soprattutto recupero delle competenze prodotte in fase di analisi.

(°) Olivetti R&S - Ivrea
(+) Istituto di Elettrotecnica ed Elettronica - Politecnico di Milano

Gli strumenti che la metodologia propone sono il linguaggio Simula67, un preprocessor (su IBM 370, scritto a sua volta in Simula67) per introdurre il concetto di modulo come strumento per la verifica delle interfacce fra unità di programmazione diverse, il Librarian per operazioni di text editing, alcuni packages di tracing, il TSO per garantire un certo grado di interattività e un insieme di procedure JCL catalogate che permettono la gestione delle librerie di progetto e la integrazione degli elementi citati.

Come prodotto parziale dell'attività di sviluppo del sistema, si sono ricavate alcune indicazioni relative all'uso del Simula67, senza altri supporti, come linguaggio di specifica eseguibile: in particolare, si sono identificate alcune modalità in base a cui strutturare programmi utilizzando sistematicamente il concetto di classe; appunto questo tema verrà trattato nel seguito.

2. CLASSI E TIPI

Il concetto di classe. La classe è il meccanismo base fornito dal Simula67 per definire tipi, cioè modelli di strutture dati unitamente alle procedure di accesso agli oggetti di ciascun tipo /1//2/.

Una istanza, cioè un esemplare, di una classe si costruisce dinamicamente con una istruzione "new".

Una operazione, definita in una classe, su una istanza della classe stessa viene richiamata prefissando il nome della operazione (ad esempio, un nome di procedura) con il nome della istanza ("dot notation"). Più in generale, la dot notation consente di accedere a qualsiasi attributo dell'oggetto. Per maggiori dettagli ed esempi, si rimanda a /2/.

Tipi user-defined e tipi astratti. La classe consente quindi di introdurre in un programma tipi definiti da un utente; una istanza di un tipo è un oggetto che può essere manipolato per mezzo delle procedure associate, senza bisogno di conoscere i suoi dettagli implementativi.

E' evidente che questa possibilità è fondamentale quando si voglia strutturare un programma per livelli di astrazione (in particolare, procedendo top-down).

Si noti che un tipo definito per mezzo del concetto di classe non è un tipo astratto in senso stretto, poichè il Simula67, nella versione attualmente distribuita, non impone che l'accesso ad un oggetto avvenga solo attraverso le procedure definite nella classe di cui è istanza. A questa limitazione si ovvierà, in APM/Simula, introducendo il concetto di "modulo" per limitare la trasparenza delle interfacce.

Concatenazione di classi. Per mezzo

del concetto di classe è possibile costruire concetti elaborati sulla base di altri più semplici.

Tale tecnica di strutturazione dei programmi va sotto il nome di concatenazione. Il risultato della concatenazione di due classi è una nuova classe, i cui attributi sono l'unione degli attributi delle due classi componenti.

Si possono rappresentare diversi livelli concettuali con una sequenza di dichiarazioni di classi, ognuna delle quali utilizza la precedente come prefisso. Ogni livello, pertanto, può utilizzare gli attributi delle classi più "esterne" come una piattaforma mentale per ulteriori costruzioni /3//4//5/.

Ad esempio, con le dichiarazioni

```
class a
   "dichiarazione attributi di a"
end;
a class b
   "dichiarazione attributi di b"
end;
```

si ha la possibilità di costruire oggetti istanze della classe b, i cui attributi, in particolare le strutture dati, sono la concatenazione di quelli definiti in a e in b, e su cui è possibile operare con le procedure definite sia in a che in b.

3. UN MODO DI STRUTTURARE PROGRAMMI

Il concetto di classe può venire utilizzato in diversi modi per ottenere programmi ben strutturati; esso tuttavia, proprio per la sua potenza, può essere fonte di confusione. In questo paragrafo si descrivono tre modi fondamentali di utilizzare questo concetto, modi che sembrano particolarmente adatti a favorire la scrittura di specifiche ben strutturate, oltre che eseguibili. Gli esempi fanno riferimento ad un esperimento di uso di APM/Simula effettuato in sede di progetto di un File System (FMS) /6/.

Mediante livelli linguistici. E' normale che un sistema software di dimensioni non piccole sia organizzato per livelli linguistici, disposti gerarchicamente. Ogni livello costituisce un "dialetto" del linguaggio base nel senso che rende disponibile un insieme di costrutti non standard, implementati ai livelli inferiori e particolarmente adatti a risolvere specifici problemi di programmazione.

Ad esempio, nel caso del FMS, si sono identificati diversi livelli linguistici, di cui i principali sono:
- il livello "applications", al quale vengono scritti i programmi applicativi, e in cui si dispone di primitive tipo "get", "put" e analoghe;
- il livello "file_system" che le implementa utilizzando primitive di livello inferiore che manipolano strutture dati organizzate ad albero;
- il livello "addressing_structure", che le implementa utilizzando meccanismi di I/O logico;

- il livello "logical_IO", che li implementa in termini di meccanismi fisici;
- il livello "physical_IO", che opera in termini di comandi di accesso a un disco fisico;
-due livelli, "interface" e "types", che verranno discussi nel seguito. Il livello base è ovviamente quello di una macchina Simula 67.

La struttura a livelli linguistici è implementata definendo ogni livello per mezzo di una classe, e concatenandole. Ad esempio:

addressing_structure class file_system

e analogamente

logical_IO class addressing_structure

eccetera.

Per limitare la visibilità degli strati interni del sistema, si impone che da un livello si possa accedere solamente alle procedure definite in quello immediatamente inferiore; ciò verrà controllato automaticamente nel sistema definitivo.

In ultima analisi, un livello linguistico è un insieme di dichiarazioni di tipi, con le operazioni associate, che serve per definire un "dialetto" di un linguaggio base orientato alla soluzione di certe classi di problemi.

Mediante contesti linguistici. Sia che il programma sia decomposto o meno in livelli, è opportuno separare l'insieme degli oggetti globali (cioè delle variabili e dei tipi accessibili in un dato punto del programma) in sottinsiemi, in base al fatto che si voglia attribuire o meno a certe classi o procedure il diritto di accedere a determinati oggetti.

Si introduce così una modularizzazione "verticale", ortogonale a quella "orizzontale" discussa in precedenza. Ciò può essere ottenuto racchiudendo i vari sottinsiemi di oggetti in classi locali a qualche livello, classi le cui istanze possono divenire contesti separati su cui operano le diverse parti del programma.

Per programmi a livelli, si può:
- introdurre un livello linguistico "types", in cui vengono dichiarati tutti i tipi user-defined pubblici. Tale livello può essere considerato una estensione del linguaggio, ed i tipi in esso definiti sono visibili ovunque:
- introdurre un livello "interface", in cui sono definite una o più classi ("contesti") che racchiudono tutte e sole le variabili non locali con la loro inizializzazione; assieme ai contesti sono definiti in "interface" gli statements che consentono di creare una istanza di ciascuno di essi.

E' appunto questo il criterio seguito in FMS; in esso, tutte le variabili non locali sono definite in un contesto appartenente al livello "interface". Ad esempio, è possibile creare come segue un contesto "c":

```
class interface;
begin
  class context;
  begin
    integer level;
    ref(type_buffer) output_buffer;
  end of context;
  ref(context) c;
  c:=new context;
end of interface;
```

Nell'esempio, "type_buffer" è un tipo user-defined, dichiarato al livello "types";"output_buffer" è una istanza di questo tipo.

Tutti i livelli linguistici al disopra di "interface" hanno accesso agli attributi "level" e "output_buffer" del contesto linguistico "c". Ad esempio se con "position(k)" si accede alla k-ma posizione di un generico oggetto di tipo "type_buffer", si può scrivere

```
x:= c.output_buffer;position(5)
```

col significato di "assegna a x il valore della posizione 5 dell'oggetto output_buffer, definito nel contesto c".

Dall'esempio appare chiaro il valore documentante della notazione; si noti che una variabile non locale è immediatamente riconoscibile, in quanto è sempre prefissata dal nome del contesto in cui è definita. Ciò introduce un fattore di sicurezza che riduce in modo significativo la possibilità di accessi non corretti.

In ultima analisi, un contesto linguistico definisce un insieme di oggetti accessibili nell'ambito di un particolare programma, o di alcuni suoi moduli specifici.

Mediante insiemi di astrazioni. Una astrazione funzionale definisce in generale una trasformazione da un insieme di oggetti di input a un insieme di oggetti di output. La maggior parte dei linguaggi di programmazione fornisce un meccanismo di tipo procedurale per implementare questi tipi di astrazioni. Un insieme di procedure, unitamente agli oggetti su cui operano, costituisce un livello linguistico.

Analogamente a quanto si è fatto, nell'ambito di un livello, per le variabili globali raggruppandole in contesti linguistici, si può pensare di voler incapsulare insiemi di astrazioni procedurali intimamente correlate fra loro (perchè operano sulla stessa, o sullo stesso insieme di, astrazioni di dati) dentro una astrazione più generale. Ad esempio:

```
class addressing_structure
  class abs_leaf;
    procedure fetch_into_buffer;
    procedure copy_from_buffer;
      :
  class abs_node;
    procedure fetch_into_buffer;
      :
  class abs_root;
    procedure copy_from_buffer;
    procedure allocate;
      :
```

Si noti che questa classe non costituisce un nuovo tipo: non contiene infatti definizioni di strutture dati.

Essa è invece una forma di "control abstraction" che è utile per ottenere, mediante la dot notation, un più elevato livello di leggibilità; si garantisce inoltre una maggiore affidabilità, evidenziando ad esempio che, se "leaf" e "node" sono istanze delle classi "abs_leaf" e "abs_node", la notazione "leaf.fetch_into_buffer" seleziona una astrazione diversa da "node.fetch_into_buffer".

4. CONCLUSIONI

Anche se taluni concetti relativi alla strutturazione dei programmi in Simula67 possono sembrare di comprensione non immediata, gli esperimenti compiuti hanno dato risultati soddisfacenti. In particolare si è ottenuta una buona strutturazione dei programmi, attraverso l'uso sistematico dei livelli linguistici, assieme ad una elevata leggibilità, attraverso l'uso della dot notation indotto dall'uso dei contesti linguistici e delle astrazioni procedurali.

I risultati ottenuti sono tali da incoraggiare il completamento di una implementazione prototipale dell'intero sistema APM/Simula, attualmente in fase di avanzato sviluppo, su cui compiere ulteriori esperienze.

Ringraziamenti. Gli autori desiderano ringraziare J. Lomas e F. Marra, che hanno realizzato il programma FMS ed a cui sono dovute numerose esperienze e suggerimenti essenziali per lo sviluppo della metodologia.

RIFERIMENTI

/1/ O.J. Dahl, B. Myhrhaug, K. Nygaard: "Simula 67 - Common Base Language", NCC pub. S22, Oct. 1970

/2/ M.G. Birtwistle et al.; SIMULA begin, Auerbach, Philadelphia, 1970

/3/ O.J. Dahl, C.A.R. Hoare: "Hierarchical Program Structures", in Structured Programming, Academic Press, London 1970

/4/ S. Brandi, T. Pedrotti: "Tipi di dati e strutture di programmi nel Simula 67", Atti AICA 1977 (vol.3), Ed. Tecnico Scientifica, Pisa 1977

/5/ S. Brandi, T. Pedrotti: "Famiglie di programmi simulatori", in corso di pubblicazione su "Rivista di Informatica".

/6/ J.P. Lomas: "User's Guide to FMS Program", Rapp. interno, Olivetti R&S (non disponibile)

IL SEGNALIBRO

In questa rubrica vengono segnalati libri, articoli o studi di recente pubblicazione che, a giudizio della redazione o dei lettori di NOTE DI SOFTWARE, rivestono un particolare interesse nell'ambito dei temi a cui questa rivista é dedicata.

Le citazioni fra virgolette, se non ne é indicata la fonte, sono tratte dai lavori stessi.

Metodologie di programmazione

● Peters,L.J., Tripp,L.L. COMPARING SOFTWARE DESIGN METHODOLOGIES, Datamation, vol. 23, n. 11 (novembre 1977), 89-94

Presenta una sintetica rassegna di alcune delle più note metodologie per la costruzione di software: Structured Design (Constantine, Myers, Yourdon, et al.), il metodo di Jackson, Logical Construction of Program (Warnier), Meta stepwise refinement (Ledgard, Schneiderman), e Higher Order Software (Hamilton e Zeldin).

● Finneran,T.R., Henry,J.S., STRUCTURED ANALYSIS FOR DATA BASE DESIGN, Datamation, vol. 23, n. 11 (novembre 1977), 99-113.

"...A structured design approach using functional analysis offers what may be the only systematic and organized method of developing the data base design. This approach is becoming widely accepted as a technique for designing application programs, and can also be used in designing the data base itself. But to apply that method, the business functions of the organization must be analyzed, not the organizational aspects of the business. [...] "

Ingegneria del software

● Ivie,E.L., THE PROGRAMMER'S WORKBENCH - A MACHINE FOR SOFTWARE DEVELOPMENT, Communications of the ACM, vol. 20, n. 10 (ottobre 1977), 746-753.

"On almost all software development projects the assumption is made that the program development function will be done on the same machine on which the eventual system will run. It is only when this production machine is unavailable or when its programming environment is totally inadequate that alternatives are considered. In this paper it is suggested that there are many other situations where it would be advantageous to separate the program development and maintenance function into a specialized computer which is dedicated to that purpose. Such a computer is here called a Programmer's Workbench. The four basic sections of the paper introduce the subject, outline the general concept, discuss areas where such an approach may prove beneficial, and describe an operational system utilizing this concept."

Documentazione

● Wright,P., PRESENTING TECHNICAL INFORMATION: A SURVEY OF RESEARCH FINDINGS, Instructional Science, vol. 6, n. 2 (aprile 1977), 93-134.

"This paper reviews research investigations into various aspects of the presentation of technical information. It considers the objectives of different readers who may be consulting the information as a reference work or who may need to assimilate the information in its entirety.

Ways of using headings, summaries and questions to achieve these differing objectives are discussed. The review also considers the usefulness of alternatives to prose, such as flow-charts, tabulation schemes and graphic presentations. It is concluded that although there is no single 'best' presentation format, and although the research literature is in many places incomplete, nevertheless there are a number of studies demonstrating the advantages of particular presentation formats in specific circumstances. The results of these studies need to be given due weighting where decisions are taken about the appropriate way of presenting any technical information."

● Wirth,N., WHAT CAN WE DO ABOUT THE UNNECESSARY DIVERSITY OF NOTATION FOR SYNTACTIC DEFINITIONS?, Communications of the ACM(Short Communications), vol. 20, n. 11 (novembre 1977), 822-823.

"... Many language definitions appear in journals, many are found in technical reports, and perhaps a greater number remains confined to proprietory circles. After frequent exposure to these definitions, one cannot fail to notice the lack of 'common denominators'. The only widely accepted fact is that the language structure is defined by a syntax. But even notation for syntactic description eludes any commonly agreed standard form, although the underlying ancestor is invariably the Backus-Naur-Form of Algol 60 report. As variations are often only slight, they become annoying for their very lack of an apparent motivation. Out of sympathy with the troubled reader who is weary of adapting to a new variant of BNF each time another language definition appears, and without any claim for originality, I venture to submit a simple notation that has proven valuable and satisfactory in use. [...]"

● Lindsey,C.H., STRUCTURE CHARTS- A STRUCTURED ALTERNATIVE TO FLOWCHARTS, SIGPLAN Notices (ACM), vol.12, n.11 (novembre 1977), 36-49.

"Conventional flowcharts are a hindrance to structured programming. An alternative notation, which emphasises the nested structure of programs, is proposed. [...] Its intention is primarily didactic. Advantages claimed over competing notations are the close correlation with the indenting of the final program, the ability to model the possibilities of expression-oriented languages, and the ability to feature just those jumps which may be considered harmless."
La notazione può anche essere utilizzata per descrivere strutture di dati, in alternativa a quella nota di Jackson.

Project Management

● Getz,C.W., MIS AND THE WAR ROOM, Datamation vol.23, n.12 (dicembre 1977), 66-70.

"... One of the effective tools of management control, and perhaps one of the least understood, is the Management Information Center, sometimes called a chart room, or situation room, or corporate war room. [....] A control room is a good mechanism to give vision to program and organization objectives. It provides a place for project teams and management to meet in an environment which created an awareness of information and time. It provides a central location where the staff should be required to portray the objectives and status information concerning their own areas of responsability. [...]"

● Fagan,M.E., INSPECTING SOFTWARE DESIGN AND CODE, Datamation, vol.23, n.10 (ottobre 1977), 133-144.

"... Design and code inspections have been applied successfully in several programming projects, both large and small, and including systems and applications programs. They have not been found to 'get in the way' of programming, but instead enabled higher predictability than other means and improved productivity and product quality. [....] Inspections are a formal, efficient, and economical method of finding errors in design and code. All instructions are examined at least once in the conduct of inspections. Key aspects of inspections are exposed in the following. [...]"

Linguaggi naturali e linguaggi di programmazione

● Wile,D., Balzer,R., Goldman,N., AUTOMATED DERIVATION OF PROGRAM CONTROL STRUCTURE FROM NATURAL LANGUAGE PROGRAM DESCRIPTIONS, Proceedings of the Symposium on Artificial Intelligence and Programming Language, Special Issue of SIGPLAN Notices, vol.12, n.8, and SIGART Newsletter, n.64 (Agosto 1977), 77-84.

"This paper describes a system which organizes a natural language description of a

program into a conventional program control structure, as a part of a larger system for converting informal natural language program specifications into running programs."

● Hobbs,J.R., WHAT THE NATURE OF NATURAL LANGUAGE TELLS US ABOUT HOW TO MAKE NATURAL--LANGUAGE-LIKE PROGRAMMING LANGUAGES MORE NATURAL, ibidem, 85-93.

"When a student is learning an algorithm from a textbook, his first approach is frequently through an English description. This is normally easier to understand than raw code, and sometimes easier than a flow-chart, in spite of the fact that programming languages are designed for algorithm specification while English is only pressed into its service. If the English is easier to understand, it is likely that it has many features that would ease programming itself. This paper investigates some of these features. [...] four aspects that caracterize coherent English texts are considered in turn: 1. redundancy, ellipsis, and contextual interpretation of words; 2. the prevalence of spatial metaphors; 3. various anaphoric devices; and 4. implicit and explicit intersentence relations. .... "

Reti di Petri

● Peterson, J.L., PETRI NETS, ACM Computing Surveys, vol.9, n.3 (settembre 1977), 223--252.

"Over the last decade, the Petri net has gained increased usage and acceptance as a basic model of asynchronous concurrent computation. This paper surveys the basic concepts and uses of Petri nets. The structure of Petri nets, their markings and execution, several examples of Petri net models of computer hardware and software, and research into the analysis of Petri nets are presented, as are the use of the reachability tree and the decidability and complexity of some Petri nets problems. Petri net languages, models of computation related to Petri nets, and some extensions and subclasses of the Petri net model are also briefly discussed."

Un nuovo libro di Hansen

● Hansen,P.B., THE ARCHITECTURE OF CONCURRENT PROGRAMS, Prentice-Hall Series in Automatic Computation, Prentice-Hall, Inc. Englewood Cliffs, New Jersey 07632, 1977, XVII+317.

Dalla prefazione: "This book describes a method for writing concurrent computer programs of high quality. It is written for professional programmers and students who are faced with the complicated task of building reliable computer operating systems or real--time control programs.
The motivations for mastering concurrent programming are both economic and intellectual. Concurrent programming makes it possible to use a computer where many things need attention at the same time - be they people at terminals or temperatures in an industrial plant. It is without doubt the most difficult form of programming.
This book presents a systematic way of developing concurrent programs in a structured language called Concurrent Pascal-the first of its kind. The use of this language is illustrated by three non-trivial concurrent programs: a single-user operating system, a job-stream system, and a real-time scheduler. All of these have been used successfully on a PDP 11/45 computer. The book includes the complete text of these three programs and explains how they are structured, programmed, tested, and described.
In an earlier book, Operating System Principles (Prentice-Hall, 1973), I tried to establish a background for studying existing operating systems in terms of basic concepts. This new text tells the other side of the story: how concurrent programs can be constructed systematically from scratch. It also illustrates details of important design problems - the management of input/output, data files, and programs - which were deliberately omitted from the first book. So it is useful both as a practical supplement to operating system courses and also as a handbook on structured concurrent programming for engineers."

Indice: PROGRAMMING TOOLS: 1.Design Principles, 2. Programming Concepts, 3. Sequential Pascal, 4. Concurrent Pascal; CONCURRENT PROGRAMS: 5. The Solo Operating System, 6. The Job Stream System, 7. A Real-Time Scheduler; LANGUAGE DETAILS: 8. Concurrent Pascal Report, 9. Concurrent Pascal Machine; THE NEXT STEP.

● Hammer, M., Howe, W.G., Kruskal, V.J., Wladawsky, I., A VERY HIGH LEVEL PROGRAMMING LANGUAGE FOR DATA PROCESSING APPLICATIONS, Communications of the ACM, vol.20, n.11 (novembre 1977), 832-840.

"Application development today is too labor intensive. In recent years, very high-level languages have been increasingly explored as a solution to this problem. The Business Definition Language (BDL) is such a language, one aimed at business data processing problems. The concept of BDL mimic those which have evolved through the years in business using manual methods. This results in three different sublanguages or components: one for defining the business form [FDC: Form Definition Component], one for describing the business organization [DFC: Document Flow Component], and one for writing calculations [DTC: Document Transformation Component]."
"... the basic data structure of BDL is the document. Documents in BDL mirror very closely those in the real world: simply data filled out on a form. Forms are the templates with which specific documents are produced. They consist of preprinted information plus specifications about the data used to fill them out. In FDC, the user describes the forms that are needed in the program. [...]
DFC is expressed in terms of a two-dimensional graphical language, which represents both the organizational units of the business and the document that flow among them. [...]
The DTC part of a program expresses the precise functional relationship among the documents of the system; it is here that the actual computations are specified. For every irreducible step occurring in the DFC, there is a corresponding section of code in DTC that describes how the output documents of the step are constructed from the inputs of the step".
Il lavoro può anche fornire interessanti spunti per problemi di documentazione EDP.