Honeywell

Honeywell Information Systems Italia

Via Laboratori Olivetti - Pregnana Milanese

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

17/18

SOFTWARE NOTE DI SISTEMA

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.
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

17/18

Maggio 1982

Indice:

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

NOTE DI SOFTWARE

QUESTO NUMERO...

Le metodologie di disegno e i linguaggi interattivi orientati all'utente sono i temi principali di questo numero doppio di NOTE DI SOFTWARE.

Il primo lavoro ha come argomento WELLMADE, la metodologia di disegno di sistemi software i cui lineamenti generali sono già stati presentati da A. Pizzarello sul numero 13/14 di NOTE DI SOFTWARE. In questo articolo, F. De Cindio, L. Pomello e S. Tessarollo discutono i vari aspetti di WELLMADE, sviluppando passo passo un semplice esempio.
Nell'articolo, gli autori evidenziano, in particolare, come il metodo e il linguaggio formale proposti da WELLMADE garantiscano la correttezza dei raffinamenti del disegno e l'intreccio costante tra costruzione del disegno del sistema e la sua documentazione.

Nel secondo lavoro, L. D'Amico presenta, con un esempio, le linee generali della nuova metodologia JSD (Jackson System Design) per il disegno di sistemi proposta da M. Jackson.
JSD, attualmente in fase di sperimentazione, comprende come sua parte integrante JSP (Jackson Structured Programming), la ben nota metodologia di programmazione che permette di derivare la struttura di un programma a partire dalla struttura dei suoi dati di ingresso e di uscita.

Nel terzo lavoro, D. Lucarella descrive sinteticamente un linguaggio interattivo progettato per l'accesso alla base di dati del sistema informativo dell'ENEL.
Tale linguaggio è stato progettato con l'obiettivo della massima semplicità d'uso.

Infine, L. Ferrario presenta una utile raccolta di "schede descrittive" di programmi scritti in Pascal, tratti dalla letteratura tecnica.

La Redazione

CONTRIBUTI TECNICI

WELLMADE PER IL DISEGNO DI UN SISTEMA "BEN FATTO"

F. De Cindio, L. Pomello, S. Tessarollo
Istituto di Cibernetica, Università degli Studi di Milano

Riassunto

La dettagliata discussione di un esempio di applicazione della metodologia WELLMADE al disegno di un semplice sistema fornisce una traccia per il suo utilizzo, evidenziando come il metodo ed il linguaggio formale che lo caratterizzano garantiscono la correttezza del disegno e l'intreccio costante tra costruzione del disegno del sistema e sua documentazione.

Abstract

An application of the methodology WELLMADE to the design of a simple system is detailedly discussed in order to provide a guideline for the use of the methodology. The presentation underlines how the method and the formal language characterizing WELLMADE grant the correctness of the design and the constant interlacing between the system design construction and its documentation.

1. INTRODUZIONE

Nell'articolo apparso sul numero 13/14 di note di Software [1], A. Pizzarello ha presentato i lineamenti generali della metodologia di disegno Wellmade, da lui stesso sviluppata in collaborazione con altri nel corso degli anni '70 presso i laboratori di ricerca della HIS.

In [1] Pizzarello ha focalizzato l'attenzione su come vengono utilizzati in Wellmade l'astrazione sui dati, che ne è il principale fondamento concettuale, ed il metodo di derivazione dei programmi corretti, proposto da Dijkstra in [2].

Questo lavoro vuole essere un contributo alla presentazione di Wellmade, che si inserisce in una più vasta attività di ricerca volta a delineare un metodo per l'analisi e il disegno di sistemi EDP [3], [4]. La dettagliata discussione dell'applicazione della metodologia al disegno di un semplice sistema si propone infatti di fornire una traccia per il suo utilizzo, dall'interpretazione dei requisiti all'ultimo livello del disegno, delineando infine le modalità per il passaggio alla fase di implementazione. Nella presentazione si è voluto evidenziare, in particolare, come il metodo ed il linguaggio formale proposti da Wellmade garantiscono la correttezza dei raffinamenti del disegno e l'intreccio costante tra costruzione del disegno del sistema e sua documentazione.

Poichè Wellmade è esplicitamente pensato per il disegno di grandi sistemi, è evidente che un così semplice sistema non esaurisce la gamma dei casi che possono presentarsi: per una trattazione completa si rimanda a [?]

Una veloce sintesi (par. 2) di quanto esposto da Pizzarello in [1], volta ad evidenziare le clausole che costituiscono la documentazione di una singola macchina astratta, è premessa alla presentazione dell'esempio (par. 3) al fine di facilitarne la comprensione.

2. COSTRUZIONE/DOCUMENTAZIONE DEL DISEGNO DI UN SISTEMA CON WELLMADE

La metodologia Wellmade si pone l'obiettivo di fornire metodi, strumenti concettuali e formali (ampiamente presentati in [1]) per l'ingegnerizzazione della fase di disegno di un sistema software di dimensioni medio/grosse tali da richiedere il coinvolgimento di più persone e quindi la sua scomposizione in più moduli.

Per minimizzare gli errori di comunicazione tra i vari progettisti coinvolti e tra questi e l'utente, dovuti all'ambiguità con cui sono abitualmente descritte le

Il presente lavoro è stato effettuato nell'ambito del Progetto Finalizzato Informatica, Sottoprogetto P1, Obiettivo METOD.

Antonio Cicu della Honeywell Information Systems Italia ha contribuito con numerosi suggerimenti ed ha reso possibile l'impiego della metodologia Wellmade sviluppata nei laboratori della Honeywell.

Fiorella De Cindio ha seguito il corso "WELLMADE/RDM" tenuto a Milano da A. Pizzarello nel giugno 1980 nell'ambito del CP Project tra HISI e Istituto di Cibernetica.

specifiche del sistema e poi quelle dei vari moduli, Wellmade affianca alla documentazione in lingua naturale (per cui comunque definisce dei vincoli sui contenuti e suggerisce la definizione di standard), l'utilizzo di un formalismo che consente una non ambigua e rigorosa costruzione e documentazione del disegno del sistema.

Per minimizzare le difficoltà della fase di integrazione dei vari moduli in cui viene scomposto il sistema, fase la cui complessità è generalmente fonte di numerosi errori e rilevanti perdite di tempo, Wellmade prevede che già nella fase di disegno del sistema, cioè già nella individuazione e costruzione dei vari moduli, siano esplicitamente progettate, documentate e verificate le modalità di una loro corretta integrazione. A questo scopo propone di utilizzare, oltre all'abituale astrazione procedurale (cioè un nome per identificare una porzione di codice con o senza parametri), l'astrazione sui dati (identificazione di tipi di dati e delle operazioni necessarie per manipolarli). Questa viene realizzata da Wellmade attraverso una gerarchia di macchine astratte, ciascuna delle quali opera su uno o più tipi di dati astratti. La definizione formale, livello per livello, della corrispondenza tra i tipi da raffinare ed i tipi che li raffinano consente una verifica rigorosa della correttezza del raffinamento.

Il primo passo del disegno consiste, per Wellmade, nel selezionare dai requisiti quelle informazioni necessarie a formalizzare nella macchina di più alto livello la descrizione della struttura complessiva del sistema. Di questa macchina:
a) si descrive in linguaggio naturale la funzionalità, le interfacce con l'utente del sistema e si precisano i vincoli a cui deve soddisfare e che ne determinano i criteri di accettazione;
b) si individuano gli *oggetti* su cui opera, definendone i *tipi* con le relative operazioni eseguibili, ed i *predicati* (eventualmente astratti) che consentono di specificare relazioni tra gli oggetti;
c) si definiscono le *specificazioni dei programmi* che operano sugli oggetti in termini di asserzioni di ingresso, di uscita e invarianti (secondo il metodo proposto da Hoare in [6]) e se ne dà il *testo* nel linguaggio dei guarded commands di Dijkstra [2], utilizzando in esso le funzioni definite sui vari tipi di dati.

Per ogni tipo di dato non descritto interamente con i tipi standard ed i costruttori di tipo messi a disposizione dal linguaggio, ma definito ricorrendo anche a componenti astratte, deve essere data una rappresentazione via via più dettagliata utilizzando, senza modificare quanto fatto in precedenza, ulteriori informazioni presenti nei requisiti del sistema

Ogni macchina di livello inferiore al primo può raffinare uno o più tipi di dati astratti definiti nei livelli superiori: la sua costruzione/documentazione è analoga a quella della macchina di primo livello, integrata con la definizione di quali tipi vengono raffinati e della rappresentazione che di essi si dà.

Il processo di costruzione/documentazione del sistema termina quando per tutti i tipi di dati astratti definiti in una delle macchine della gerarchia è definita una rappresentazione nei termini dei tipi standard e dei costruttori di tipo, che tenga conto di eventuali vincoli hardware e software imposti dall'ambiente in cui il sistema deve operare e consenta di realizzare come programmi le funzioni su di essi definite.

La documentazione di una singola macchina (fig. 1) è

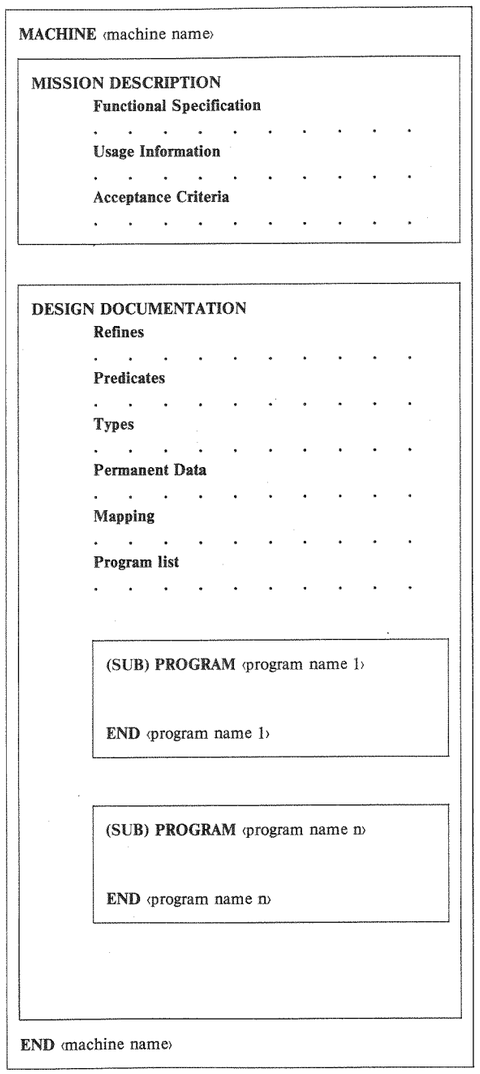


Fig. 1: La struttura della documentazione della i-esima macchina.

costituita da due sezioni: la **mission description** e la **design documentation,** che costituiscono rispettivamente la descrizione del disegno del sistema in linguaggio naturale e la sua formalizzazione nel linguaggio definito da Wellmade.

Poichè la metodologia enfatizza gli aspetti formali e sottolinea i pericoli di ambiguità insiti nel linguaggio naturale, nel seguito della **mission description** sono evidenziati solo i tratti salienti ed essa verrà per brevità tralasciata nell'esempio.

La **mission description** fornisce una descrizione di massima del sistema in linguaggio naturale. È costituita da:

– **functional description:** fornisce indicazioni sulle decisioni di disegno del sistema incorporate nella macchina (obbiettivi, funzionalità, risultati, condizioni d'uso).

– **usage information:** descrive le modalità d'uso della macchina (linguaggio, interface, supporti). Nel caso che questa presenti una interfaccia diretta con l'utente, le informazioni in essa contenute costituiscono la base per lo USER MANUAL (inteso sia come OPERATOR MANUAL che come END-USER MANUAL). Per questo motivo deve essere il più possibile completa, chiara e ben organizzata ed elementi di standardizzazione possono essere utili.

– **acceptance criteria:** indica quali vincoli (limiti di utilizzo delle risorse, richieste di prestazioni, altri vincoli hardware, riutilizzo di software già disponibile, ecc.) l'implementazione della macchina deve soddisfare affinchè lo siano quelli complessivi richiesti al sistema.

La **design documentation** fornisce la descrizione formale del disegno del sistema e costituisce il cuore della documentazione; è costituita da sei clausole per ciascuna delle quali è definita la sintassi.

– **Types** dove vengono definiti i tipi utilizzati dalla macchina. Ad ognuno è attribuito un nome a cui viene associata la struttura utilizzando, per descrivere quest'ultima, i tipi standard e i costruttori di tipo messi a disposizione dal linguaggio (cfr. fig. 2, DATA DESCRIPTION LANGUAGE punti 1 e 2).

Se, una volta identificato un tipo e assegnatogli un nome, si vuole demandare la descrizione della sua struttura ad un'altra macchina, lo si dichiara **abstract.** In questo caso, proprietà invarianti e operazioni sul tipo astratto o su tipi che eventualmente lo contengono come componente sono definite secondo lo schema (leggermente semplificato rispetto a quello dato in fig. 2 DATA DESCRIPTION LANGUAGE, punto 5, dove è prevista la possibilità di tipi parametrici):

‹type name›: ‹type expression›
let ‹data declaration›
[**inv** ‹invariant assertion›]
ofun (oppure **vfun, init**) ‹function name›
[(‹input parameters›)]
[**returns** (‹output parameters›)]
pre ‹assertion›
post ‹assertion›
end ‹type name›

dove:

- ‹data declaration› è la dichiarazione di uno o più varabili locali che servono per la specificazione delle asserzioni;
- ‹invariant assertion› indica un'eventuale limitazione sui valori delle variabili di quel tipo;
- **ofun** indica un'operazione che comporta un cambiamento del valore dell'oggetto a cui si applica e non ha necessariamente valori di ritorno;
- **vfun** indica un'operazione che non comporta un cambiamento del valore dell'oggetto a cui si applica e deve avere valori di ritorno;
- **init** specifica le operazioni che inizializzano gli oggetti del tipo;
- **pre** ‹assertion› e **post** «assertion» specificano la funzionalità di ogni funzione definendo lo stato delle variabili prima e dopo l'applicazione della funzione stessa.

– **Permanent data:** fornisce la lista di tutte le variabili che sono disponibili a tutti i programmi della macchina, indicandone per ciascuna il tipo.

– **Program list:** lista dei nomi dei programmi della macchina e loro documentazione. Per ogni programma la documentazione prevede:

- una descrizione in lingua naturale delle funzionalità che il programma deve fornire **(overview);**
- l'elenco dei nomi dei sottoprogrammi chiamati **(subprograms);**
- l'elenco delle variabili locali al programma **(variables)** che sono quindi elaborabili solo nel programma stesso e nei suoi sottoprogrammi;
- la descrizione dello stato delle variabili prima e dopo l'esecuzione e la definizione, per ogni iterazione contenuta nell'algoritmo, dell'asserzione invariante e della funzione che ne controlla la terminazione **(Specifications,** rispettivamente: **precondition, postcondition, loop invariant, variant function);**
- il testo del programma scritto nella cosiddetta **P-notation,** ossia utilizzando il linguaggio dei guarded commands di Dijkstra [2] (*); ogni programma può utilizzare le funzioni **(ofun, vfun, init)** definite sui tipi astratti della macchina e, a meno che la macchina non sia la prima, è la realizzazione di una funzione su tipi astratti che la macchina raffina. In questo caso il nome del programma è lo stesso della funzione corrispondente e le pre e post-condizioni sono direttamente ricavate dalle pre e post-condizioni della funzione.

– **Predicates:** vengono qui elencati e specificati i predicati simbolici che sono utilizzati nella macchina su dati astratti e/o per alleggerire il formalismo delle varie clausole con l'uso di nomi evocativi secondo lo schema:

‹data declaration›
‹predicate› **means** ‹text›
[**characterized by** ‹properties›]

dove:

- ‹data declaration› è la dichiarazione di uno o più variabili necessarie a definire a quali oggetti si applicano i predicati elencati ;
- ‹predicate› è l'identificatore del predicato;
- ‹text› può specificare il significato del predicato simbolico per mezzo di un'espressione logica costituita in termini dei tipi di dati elementari oppure consiste della sola parola chiave **abstract** per indicare che si tratta di un predicato definito su un tipo a questo livello della gerarchia non ulteriormente specificato. In questo caso la individuazione delle proprietà che lo caratterizzano **(characterized by** ‹properties›) consente di definire il predicato prescindendo dalla rappresentazione del corrispondente tipo.

– **Refines** compare solo nelle macchine di livello inferiore al primo e consta dell'elenco dei tipi definiti in altre macchine **abstract** o con componenti **abstract** di cui la macchina in questione realizza una rappresentazione.

– **Mapping** definisce la corrispondenza tra gli oggetti ed i predicati della macchina e quelli elencati nella clausola **Refines.**

Infatti, scendendo di livello nella gerarchia, ogni macchina realizza un tipo (o più tipi) definito precedentemente **abstract,** o composto di componenti di cui almeno una **abstract,** specificandone una rappresentazione che consenta di realizzare come programmi le funzioni ad esso associate.

(*) Una breve rassegna del linguaggio è data in figura 2 in PROGRAM DESCRIPTION LANGUAGE.

La corrispondenza tra il tipo da raffinare (indicato nella clausola **Refines**) ed il tipo che lo raffina (indicato nella clausola **Types)** viene così "scomposta" nella corrispondenza tra gli oggetti (su cui operano i programmi) e nella corrispondenza tra i predicati (che ne consentono la definizione delle specificazioni) definiti nelle due macchine secondo lo schema:

let ‹data declaration›
‹clause-1›
. . . . .
‹clause-n›

dove la generica ‹clause-i› è della forma:

‹expression-i_1› **represents** ‹expression-i_2›

oppure della forma:

‹assertion-i_1› **represents** ‹assertion-i_2›.

Nel primo caso, si definisce una corrispondenza tra gli oggetti propri della macchina (‹expressions-i›), definiti, separatamente o come componenti, nella clausola **Permanent data** e gli oggetti virtuali del tipo da raffinare (‹expression-i_2›) definiti localmente (‹data declaration›).

Nel secondo caso, si individua quale predicato della macchina di livello inferiore rappresenta un predicato definito nella macchina di livello superiore, entrambi applicati ad oggetti di cui una precedente clausola ha già definito la corrispondenza.

La definizione della corrispondenza tra il tipo da raffinare ed il tipo che lo raffina consente una *verifica formale della correttezza del raffinamento* che consiste nel verificare che le proprietà fissate nella macchina di livello superiore su di un tipo (**type invariant** e proprietà dei predicati su di esso definiti) siano soddisfatte dal raffinamento prescelto.

Un esempio non interpretato di un possibile sviluppo del disegno di un sistema in una gerarchia di macchine è presentato in fig. 3. In esso la macchina M2 raffina il tipo t_{11} di M1 e realizza nei programmi p_{21} e p_{22} (che richiamano entrambi il sottoprogramma p_{23}) le funzioni f_{11} e f_{12} definite su t_{11} in M1. La macchina M3 raffina il tipo t_{13} definito in M1. La macchina M4 raffina i tipi t_{12}, t_{22}, t_3 e realizza nel programma p_4 la funzione f_3 definita su t_3 in M3, mentre M5 raffina il solo tipo t_{21} definito in M2. Le freccie che collegano le varie macchine ne visualizzano le dipendenze gerarchiche; ad esempio evidenziano che M4 è gerarchicamente dipendente da M1, M2 e M3, M5 lo è da M2, mentre M4 e M5 sono reciprocamente indipendenti.

NOTATION CONVENTIONS
O,A,B,X,Y are object symbols
t is a type symbol
F is a function symbol
V denotes a value
{See "Dot and Colon Notation"}
NOTE: In addition, the colon (:) is used to separate a symbol and its definition
{See "Type Declarations" and "Data (Variable) Declarations"}
[ ] information between brackets is optional
‹ › denotes a syntactic symbol
{} denotes explanatory information
‹text› denotes English text

DATA DESCRIPTION LANGUAGE
1. STANDARD DATA TYPES
and their operations

a. integer
{unary} -, {unary} +,
+, -, *, :, =, ≠, ‹, ›, ≤, ≥, :=

b. real
{unary} -, {unary} +,
+, -, *, /, =, ≠, ‹, ›, ≤, ≥, :=

c. boolean
{constant} T, {constant} F
not, and, or, xor, cand, cor, :=, =, ≠

d. char
=, ≠, :=

e. (lv...hv) {subrange}
where lv and hv are lowest value and highest value of a primitive type

f. (V1[]V2[]...[]Vn) {enumerated}
where V1, V2, ..., Vn are the only permissible values of this data type
(eg., COLOR: (RED[]WHITE[]BLUE)

g. **abstract** {a type designator whose definition is given elsewhere}

h. **given** {a type designator for existing functionalities}

2. **COMPOSITE DATA TYPES**
and their operations

a. **Cartesian Product** {Concatenation}
ordered (01:t1; 02:t2; ...;0n:tn)
unordered (01: t1, 02:t2, ...,0n:tn)
selector operation:
if A is an ordered or unordered cartesian product, then A.0i or A:0i selects the ith component

b. Discriminated Union {Alternated Data Type}
(V1:t1[]V2:t2[]...[]Vn:tn)
tag operation:
if A is a discriminated union, then the variable A.tag has the value Vi if the value of A is of type ti

c. **Array** {or sequence}
t **array**
Operations
If A is an array of items of type t then
A.lob - the integer which is the smallest index
A.hib - the integer which is the largest index
A.dom - the non-negative integer which is the number of elements:
A.dom=A.hib-A.lob+ 1≥0
A(I) - the value of the Ith item
A.low - the first item
A.high - the last item
A:lorem - remove the first item
A:hirem - remove the last item
A:loext(a) - append a new first item, a
A:hiext(a) - append a new last item, a
A:=(k,a1,a2,an) - initialize A with
A.lob=k, and A (k)=al.
A (k+1)=a2,...,A (k+n-1)=an
A:swap(I,J)-exchange the Ith and Jth item
A.inc.‹index›- increase the index to the next item
A.dec.‹index›- decrease the index to the previous item

3. **Dot and Colon Notation**
. - Object.function (parameters) value
producing operation on a data structure
Eg.: A.dom, A.lob, B.tag
: - Object:function (parameters) value
changing operation on a data structure
Eg.: A:loext, A:swap(3,4)

4. **Data (Variable) Declarations Format**
‹object id›:‹type expression›
{‹text›}
Eg.: X:**integer** [a whole number]

5. **Type Declarations**
‹type name›[(‹parameter list›)]:‹type expression›:
[let‹data declarations›]
[inv‹invariant assertion›]
‹designator›‹function name›[(‹parameters›)]
[returns(‹parameters›)]
[pre‹specifications›]
[post‹specifications›]
.
.
[end‹type name›]
where ‹designators› is **ofun** to designate a function which change the object value, **vfun** for a function which does not change the object value, and **init** for a function which initialize the object. ‹parameter list›, ‹data declarations›, and ‹function parameters› is an unordered cartesian product, ‹type expression› is an expression describing a composite type or a standard type or a name or any declared type. ‹invariant assertion› and ‹specifications› are assertions.
Eg.:
Stack(n:integer) :name **array**
let S:stack(n) , n: integer
inv S.dom‹n
ofun push (X:name)
pre :S.dom≤n,
post S'.dom = S.dom + 1 **and**
S'.high = X

vfun full **returns** (b:boolean)
post (S.dom‹n=› b=true)
or
(S.dom›n =› b=false)
end stack

PROGRAM DESCRIPTION LANGUAGE
1. **Sequential Operations**
S1; S2 {Si-statements}
2. **No Operation**
skip
3. **Assignment**
X1, X2, ... Xn:=E1, E2, ..., En
The value of the expression, Ei, is assigned to be the value of the variable, Xi. Assignement is made in parallel to all Xi, i=1,2, ...n

Fig. 2: "Quick reference guide" alla notazione di WELLMADE (tratta da [3])

4. **Selection**
if B1—›S1—›[]B2—›S2[]...[]Bn—›Sn **fi**
where B1, B2, ..., Bn are boolean-valued expressions called the guards
and S1, S2, ..., Sn are program statement lists called the guarded commands
A true guard, Bi, is selected and the corresponding guarded command Si is executed. At least one guard must be true, else the program aborts.

5. **Iteration**
do B1—›S1—›[]B2—›S2[]...[]Bn—›Sn **od**
where B1, B2, ..., Bn are boolean-valued expressions called the guards
and S1, S2, ..., Sn are program statement lists called the guarded commands
A true guard, Bi, is selected, the corresponding guarded command is executed and then the process repeated. When all guards are false, the statement has no effect.

6. **Functions**
Data structure operations
infix (x + y)
dot A.low {a value is returned which may be assigned}
colon A:hiext(I) {the value of A is changed; a value may or may not be returned}

Programs
‹program name›[(‹actual parameters›)]
[returns(‹actual parameters›)]

7. Comments
{‹text›}

Mapping
[docal object declaration›;]
‹expression1›**represents**‹expression2›;
‹assertion1›**represents**‹assertion2›;
‹expression1›and‹assertion1› are relative to objects of this machine. ‹expression2› and ‹assertion2› are relative to objects of the types of upper level machines refined here and declared as local objects.

SPECIFICATIONS
Notation for assertions, invariants, requirements, conditions, and input, output, or state specifications.

FORMS
1. **English text** {‹text›}
2. **Predicate Definitions:**
‹local object declaration›;
‹predicate›**means**‹text›;
‹predicate›**characterized by**‹properties›;
1. **Assertion Syntax**
a. Logical connectors
or (V), **and** (∧,&), **not**(non,¬), **if ... then implies**(=›) ,**iff**(‹=›)
b. Quantifiers
existential - (∃,**E, there exists, for some)**
universal - (∀,**A, for all**)
c. Range of quantified variables can be expressed by using **in** followed by a dom function or the range in parenthesis.
Eg.: **for all** i **in** A.dom
for all j **in** (A.lob....h)
d. All predicate terms are enclosed in square brackets, for example:
for all i[(A.lob≤i≤A.hib]
=› [POP(x(i), X' (i))]
e. The notation:
$ {S/P(S)}
indicates the number of elements of the set S having the property P(S).
f. Primed symbols denote values after the execution of the program; wheres non-primed symbols denote values before execution.

REQUIREMENTS INTERPRETATIONS ‹name›:
‹text›
SYSTEM‹name›
‹machine name›{‹text›}
‹machine name›{‹text›}
.
.
‹machine name›{‹text›}
MACHINE‹machine id›:
MISSION DESCRIPTION
Functional Specification
‹text›
Usage Information
{‹text›}
Acceptance Criteria
{‹text›}
.
DESIGN DOCUMENTATION
Refines‹type name list›
Predicates
‹predicate descriptions›;
.
.
‹predicate descriptions›
Types
‹type declarations›;
.
.
‹type declarations›
Data
‹data declarations›;
.
.
‹data declarations›
[inv ‹assertion›]
Mapping‹representation function›
Programs
‹program name›{‹text›},
.
.
‹program name›{‹text›}
init ‹program name› {‹text›}
PROGRAM‹program name› (‹parameter list›)
returns‹parameter›
Overview
‹text›
Variables
‹variable declaration›;
.
.
‹variable declaration›
Subprograms
‹program name›{‹text›},
.
.
‹program name›{‹text›}
Specifications
precondition{‹assertion›}
postcondition{‹assertion›}
performance{‹assertion›}
loop invariants‹assertion›
P-notation
‹program text›[**VERSION**‹version name›]
Notes
[‹text›]
END ‹program name›
PROGRAM‹program name›(‹parameter list›)
END‹machine name›

END‹system name›

Fig. 2 (continuazione)

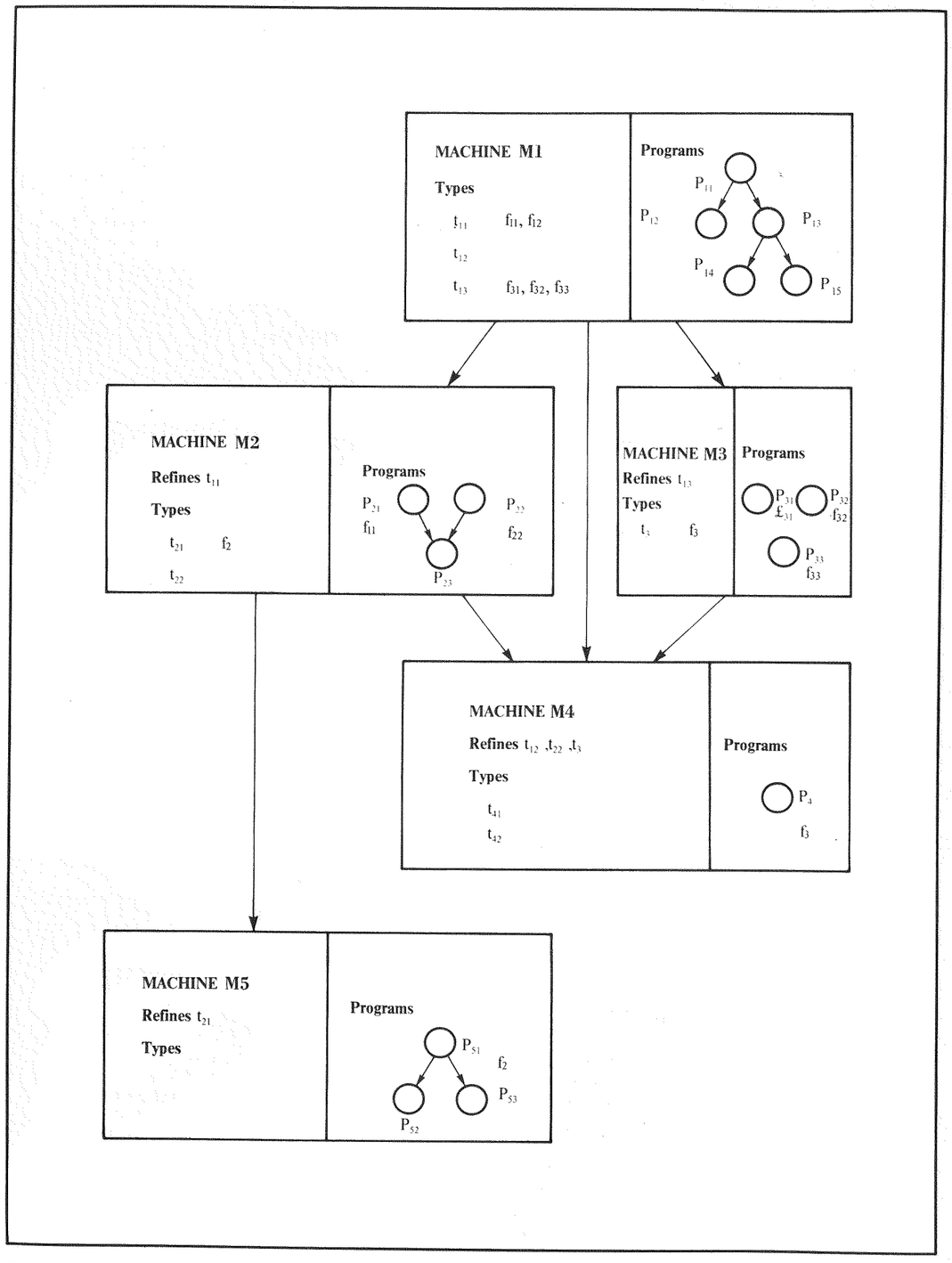


Fig. 3: un esempio non interpretato di gerarchia di macchina WELLMADE

3. IL SISTEMA "AGGIORNAMENTO CONTI CORRENTI"

3.1 Le specifiche del problema

Il sistema deve risolvere il seguente problema (tratto da [7], pag. 150, n. 5): sulla base di informazioni relative a operazioni bancarie contenute in un archivio di transazioni, deve essere aggiornato un archivio di conti correnti e stampata una serie di informazioni.

Gli elementi di ciascuno dei due archivi sono così definiti:

– un singolo conto corrente consiste del numero di conto e del saldo relativo;
– una singola transazione consiste del numero di conto a cui la transazione di riferisce, di un'informazione che specifica se si tratta di un deposito oppure di un prelievo, e dell'ammontare della transazione stessa.

Per ogni conto corrente deve essere stampata una riga contenente il numero di conto, il saldo iniziale, il numero di depositi, il numero di prelievi, il saldo finale.

L'hardware prescelto è l'Honeywell L62, release software 6.10; il linguaggio di programmazione è il PASCAL, secondo lo standard fissato in [8].

3.2 La macchina di primo livello (M1)

Il primo passo da compiere è individuare nei requisiti quelle informazioni che servono, ad un primo livello di astrazione, a delineare la struttura dell'intero sistema. In particolare, si devono anzitutto identificare gli oggetti su cui questo opera ed il loro tipo; in questo caso sono:

- un archivio di conti correnti
- un archivio di transazioni

da cui si deve ricavare

- una sucessione di resoconti.

Si tratta quindi, nei tre casi, di una sucessione di elementi omogenei (records), ciascuno contenente il numero di conto corrente ed altri dati ad esso specifici. Queste informazioni possono essere così formalizzate:

Types
account-file: account **array**
account: (anum: num; ainfo: aaltro)
aaltro: **abstract**

transaction-file: transaction **array**
transaction: (tnum: num; tinfo: taltro)
taltro: **abstract**

report-file: report **array**
report: (rnum: num; rinfo: raltro)
raltro: **abstract**

num: **abstract**

Permanent data
af: account-file
tf: transaction-file
rf: report-file

Individuati tipi e oggetti, dai requisiti si devono ora selezionare le informazioni riguardanti la loro elaborazione, da formalizzare nella definizione delle funzioni che operano sugli oggetti dei vari tipi. Verrà qui fatto uso di predicati astratti la cui caratterizzazione costituirà l'apposita sezione **Predicates.**

Qualunque sia l'algoritmo scelto per l'aggiornamento dell'archivio dei conti correnti e per la creazione del resoconto, esso implica la ricerca negli archivi conti correnti e transazioni (af e tf) di elementi relativi allo stesso numero di conto. Sono quindi necessarie funzioni a valore booleano per confrontare due oggetti di tipo num.

La formalizzazione del tipo num è quindi così completata (si ricordi che se x è una variabile di un qualunque tipo, Wellmade conviene di indicare con x' la variabile stessa dopo l'esecuzione di una funzione o di un programma):

num: **abstract**
let n, n' : num
inv (n $\circledcirc$ n') $\Longleftrightarrow$ (n $\textcircled{<}$ n' **or** n' $\textcircled{<}$ n)
vfun iseq (m: num) **returns** (b: boolean)
pre: true
prost: b = true $\Longleftrightarrow$ n $\ominus$ m
vfun ismin (m:num) **returns** (b:boolean)
pre: true
post: b = true $\Longleftrightarrow$ n $\textcircled{<}$ m
vfun ismaj (m:num) **returns** (b:boolean)
pre: true
post: b = true $\Longleftrightarrow$ n $\textcircled{>}$ m
end num

Benchè non precisato dalle specifiche del problema, è ragionevole supporre che l'archivio dei conti correnti sia ordinato secondo il numero di conto e che esista un solo record relativo ad uno stesso numero di conto. Queste ipotesi vengono formalizzate nell'invariante sul tipo account-file:

account-file: account **array**
let A: account-file
inv (A) strettamente-progressivo-su-anum
end account-file

L'ipotesi di ordinamento non è invece possibile sull'archivio delle transazioni, poichè è sensato pensare che queste si susseguano secondo l'ordine in cui realmente sono state effettuate agli sportelli della banca. Tuttavia al fine di ridurre in fase di esecuzione le scansioni degli archivi, si vuole che anche il file delle transazioni venga ordinato prima di procedere all'aggiornamento dei dati e alla produzione dei resoconti. Si definisce a questo scopo la funzione *sort* sul tipo transaction-file che viene

MACHINE M1

MISSION DESCRIPTION

. . . . . .

DESIGN DOCUMENTATION

Predicates

let n1, n2, n3: num;
n1 ⟨ n2 **means abstract**
n1 ⩵ n2 **means abstract**
characterized by
(p1) (**not** n1 ⟨ n1) **and**
(p2) (n1 ⟨ n2 ⟹ **not** n2 ⟨ n1) **and**
(p3) ((n1 ⟨ n2 **and** n2 ⟨ n3) ⟹ n1 ⟨ n3) **and**
(p4) ; (n1 ⩵ n1) **and**
(p5) (n1 ⩵ n2 ⟹ n2 ⩵ n1) **and**
(p6) ((n1 ⩵ n2 **and** n2 ⩵ n3) ⟹ n1 ⩵ n3) **and**
(p7) (n1 ⩵ n2 ⟹ (**not** n1 ⟨ n2 **and not** n2 ⟨ n1))
n1 ⩽ n2 **means** (n1 ⟨ n2 **or** n1 ⩵ n2)
n1 ≠ n2 **means not** n1 ⩵ n2
n1 ⟩ n2 **means** (**not** n1 ⩵ n2 **and not** n1 ⟨ n2)
let A:account-file ; a,a':account ;
T,T':transaction-file ; t,t':transaction;
R:report-file ; r,r':report ;
n:num ; i,j:integer;
TFinfo(n) = { T(i).tinfo / i **in** T.dom **and** T(i).tnum = n}
(A) strettamente-progressivo-su-anum **means**
(A.lob‹i‹A.hib ⟹ A(i).anum ⟨ A(i+1).anum)
(R) strettamente-sequenziale-su-rnum **means**
(R.lob ‹i‹R.hib ⟹ R(i).rnum ⟨ R(i+1).rnum)
(T) ordinato-su-tnum **means**
(T.lob‹i‹T.hib ⟹ T(i).tnum ⩽ T(i+1).tnum)
(T') esaurisce (T) **means**
(∀i **in** T'.dom ∃ j **in** T.dom: (T'(i).tnum ⩵ T(j).tnum
and (T'(i).tinfo) uguale-a (T(j).tinfo)))
(a'.ainfo) coincide-con (a.ainfo) **means abstract**
(t'.tinfo) uguale-a (t.tinfo) **means abstract**
(a.ainfo) aggiornato-con (t.tinfo) in (a'.ainfo) **means abstract**
(r.rinfo) creato-con (a.ainfo) **means abstract**
(r.rinfo) modificato-con (t.tinfo) in (r'.rinfo) **means abstract**
(r.rinfo) costruito-con (a.ainfo,TFinfo(n)) **means**
((r.rinfo) creato-con (a.ainfo) **and**
∀i **in** t.dom: (T(i).tnum ⩵ a.anum ⟹
(r.rinfo) modificato-con (T(i).tinfo) in (r'.rinfo)))
(r.rinfo) è-stand-msg **means abstract**
(r.rinfo) è-error-msg **means abstract**
characterized by . . . . . .

Types

account-file: account **array** :
let A: account-file
inv (A) strettamente-progressive-su-anum
end account-file
account: (anum: num ; ainfo: aaltro) :
let a: account
ofun update (t:transaction)
pre: a.anum ⩵ t.tnum
post: (a.ainfo) aggiornato-con (t.tinfo) in (a'.ainfo)
end account
aaltro: **abstract**

transaction-file: transaction **array** :
let T: transaction-file
ofun sort
pre: true
post: (T') ordinato-su-tnum **and** (T') esaurisce (T)
end transaction-file
transaction: (tnum: num ; tinfo: taltro)
taltro: **abstract**

report-file: report **array** :
let R: report-file
inv (R) strettamente-sequenziale-su-rnum
end report-file
report: (rnum: num ; rinfo: raltro) :
let r: report
ofun create (a:account)
pre: true
post: r'.rnum ⩵ a.anum **and** (r'.rinfo) è-stand-msg
and (r'.rinfo) creato-con (a.ainfo)
ofun modify (t:transaction)
pre: r.rnum ⩵ t.tnum **and** (r.rinfo) è-stand-msg
post: (r.rinfo) modificato-con (t.tinfo) in (r'.rinfo)
ofun error
pre: true
post: r'.rnum ⩵ t.tnum **and** (r'rinfo) è-error-msg
end report

num: **abstract** :
let n.n': num
inv (n ≠ n') ⟺ (n ⟨ n' **or** n' ⟨ n)
vfun iseq (m: num) **returns** (b:boolean)
pre: true
post: b = true ⟺ n ⩵ m
vfun ismin (m:num) **returns** (b:boolean)
pre: true
post: b = true ⟺ n ⩽ m
vfun ismaj (m:num) **returns** (b:boolean)
pre: true
post: b = true ⟺ n ⩾ m
end num

Permanent data

af: account-file ;
tf: transaction-file ;
rf: report-file

Fig. 4: La macchina di primo livello M1

PROGRAM current-accounts-updatign

Variables

r: report ; i,j: integer

Specifications

{sia TFinfo(n:num) = { tf(i).tinfo / i **in** tf.dom **and** tf(i).tnum ⩵ n}}
precondition
(af) strettamente-progressivo-su anum **and** tf.dom › 0 **and** rf.dom = 0
postcondition
(tf') ordinato-su-tnum **and** (tf') esaurisce (tf) **and** af'.dom = af.dom **and**
∀j **in** af'.dom: af'(j).anum ⩵ af(j).anum **and**
[(∀i **in** tf.dom: tf(i).tnum ≠ af(j).anum ⟹ (af'(j).ainfo) coincide-con (af(j).ainfo)) **and**
(∃i **in** tf.dom: tf(i).tnum ⩵ af(j).anum ⟹
∀i **in** tf.dom (tf(i).tnum ⩵ af(j).anum): (af(j).ainfo) aggiornato-con (tf(i).tinfo) in (af'(j).ainfo))] **and**
∀k **in** rf'.dom: [(∃j **in** af.dom ∃i **in** tf.dom: rf'(k).rnum ⩵ af(j).anum ⩵ tf(i).tnum ⟹
((rf'(k).rinfo) è-stand-msg **and** (rf'(k).rinfo) costruito-con (af(j).ainfo,TFinfo(rf'(k).rnum))
and ((∃i **in** tf.dom: rf'(k).rnum ⩵ tf(i).tnum **and** ∀j **in** af.dom: rf'(k).rnum ≠ af(j).anum) ⟹
(rf'(k).rinfo)è-error-msg)]
loop invariant
{sia Ni l'insieme dei numeri di conto distinti presenti in tf fino all'i-esimo elemento escluso)
(tf') ordinato-su-tnum **and** (tf') esaurisce (tf) **and**
∀h (af.lob‹h‹j): af'(h).anum ⩵ af(h).anum **and**
[(∀n **in** Ni: n ≠ af(h).anum ⟹ (af'(h).ainfo) coincide-con (af(h).ainfo)) **and**
(∃ n **in** Ni: n ⩵ af(h).anum ⟹
∀i **in** tf.dom (tf(i).tnum = af(j).anum): (af(h).ainfo)aggiornato-con (tf(i).tinfo) in (af'(h).ainfo))] **and**
∀k **in** rf'.dom [(∃h(af.lob‹h‹j) n **in** Ni: rf'(k).rnum ⩵ af(h).anum ⩵ n ⟹
((rf'(k).rinfo) è-stand-msg **and** (rf'(k).rinfo) costruito-con (af(h).ainfo,TFinfo(rf(k).rnum)))**and**
((∃n **in** Ni: rf'(k).rnum ⩵ n **and** ∀h (af.lob h‹j): rf'(k).rnum ≠ af'(h).anum) ⟹
(rf'(k).rinfo) è-error-msg)]
variant function
y = tf.hib - 1

P-notation

tf:sort;
j:=af.lob ; i:=tf.lob ; rf:=(1) ;
do i‹tf.hib ⟶
do j‹af.hib **cand** af(j).anum.ismin(tf(i).tnum) ⟶ j:=j+1 **od;**
if j‹af.hib **cand** af(j).anum.iseq(tf(i).tnum) ⟶
r:create(af(j));
do i‹tf.hib **cand** af(j).anum.iseq(tf(i).tnum) ⟶
af(j):update(tf(i));
r:modify(tf(i));
i:=i+1
od;
rf:hiest(r) ; j:=j+1
□ j‹af.hib **cand** af(j).anum.ismaj(tf(i).tnum) ⟶
r:error(tf(i));
rf:hiext(r);
i:=i+1;
do i‹tf.hib **cand** rf.high.rnum.iseq(tf(i).tnum) ⟶ i:=i+1 **od**
□ j‹af.hib ⟶ **do** i‹tf.hib ⟶ r:error(tf(i));
rf:hiext(r);
i:=i+1;
do (i‹tf.hib **cand** rf.high.rnum.iseq(tf(i).tnum)) ⟶ i:=i+1 **od**
od
fi
od

Fig. 4 (continuazione)

così completato:

```
transaction-file: transaction array
    let T: transaction-file
    ofun sort
        pre: true
        post: (T') ordinato-su-tnum and
              (T') esaurisce (T)
end transaction-file
```

I requisiti del sistema richiedono che ogni transazione dia luogo ad un aggiornamento del corrispondente conto corrente. È perciò necessario introdurre una funzione *update* sul tipo account che ne completa la definizione:

```
account: (anum: num ; ainfo: aaltro)
    let a: account
    ofun update (t: transaction)
        pre: a.num = t. tnum
        post: (a.ainfo) aggiornato-con (t.tinfo) in
              (a'.ainfo)
end account
```

Infine, per completare la definizione delle funzioni sui vari tipi, bisogna tenere presente che ogni riga di resoconto, che si riferisce ad un differente numero di conto, viene creata e successivamente modificata se più transazioni si riferiscono allo stesso numero di conto ed infine accodata a quelle già prodotte. Sul tipo report-file quindi non è necessario definire alcuna funzione in quanto basta utilizzare la funzione standard *hiext,* ma è opportuno formalizzare in un invariante il fatto che un oggetto di tipo report-file, per come viene costruito, risulta anch'esso ordinato sui numeri di conto senza ripetizioni.
Quindi si ha:

```
report-file: report array
    let R: report-file
    inv (R) strettamente-sequenziale-su-rnum
end report-file
```

Sono invece necessarie due funzioni del tipo report: una (**ofun** create) per creare un nuovo elemento di questo tipo (che verrà accodato agli altri con la *hiext*) e una per poter aggiornare un singolo report quando più transazioni si riferiscono allo stesso numero di conto (**ofun** modify). È qui però necessario tenere presente che tali funzioni possono essere utilizzate solo quando per una o più transazioni relative ad un dato numero di conto esiste il corrispondente sull'archivio dei conti correnti. Quando tale corrispondente non esiste è necessario invece segnalare una situazione di errore con una opportuna funzione (**ofun** error), che crea un messaggio di errore, che verrà accodato agli altri resoconti con la *hiext*. Per caratterizzare queste due differenti situazioni si utilizzano due predicati sulla componente rinfo del tipo report; rispettivamente "è-stand-msg" e "è-error-msg". Formalmente si ha:

```
report: (rnum: num ; rinfo: raltro)
    let r: report
    ofun create (a: account)
        pre: true
        post: r'.rnum ∈ a.anum and
              (r'.rinfo) è-stand-msg and
              (r'.rinfo) creato-con (a.ainfo)
    ofun modify (t: transaction)
        pre: r.rnum ∈ t.tnum and
             (r.rinfo) è-stand-msg
        post: (r.rinfo) modificato-con (t.tinfo)
              in (r'.rinfo)
    ofun error
        pre: true
        post: r'.rnum ∈ t.tnum and
              (r'rinfo) è-error-msg
end report
```

Per quanto concerne la stesura di programmi, Wellmade fa proprio l'approccio costruttivo proposto da Dijkstra in [2], che consiste nella preliminare formalizzazione delle ipotesi sotto cui ogni programma opera in una asserzione logica sui suoi dati di ingresso, detta precondizione, e delle funzionalità che realizza nella cosidetta postcondizione. L'algoritmo scelto è caratterizzato da una asserzione invariante per ciascuna iterazione presente in esso.

Le specificazioni dei programmi così formalizzate, unite alle specificazioni delle funzioni che operano sui vari tipi di dati servono da guida alla corretta costruzione del codice.

Nel nostro caso, la precondizione deve indicare che il programma è progettato per operare correttamente nell'ipotesi che l'archivio af dei conti correnti sia ordinato; che l'archivio tf delle transazioni contenga almeno un elemento; che la successione dei resoconti rf sia inizialmente vuota. Poichè la prima di queste condizioni è garantita dall'invariante sul tipo account-file, la precondizione del programma è:

tf.dom>0 **and** rf.dom = 0

Per quanto riguarda la postcondizione, valgono le seguenti considerazioni che consentono di costruirla precisando lo stato dei tre archivi dopo l'esecuzione del programma.

– Su tf: le transazioni non vengono introdotte o cancellate o modificate con nessuna funzione, ma tf deve anzitutto venire ordinato mediante l'esecuzione della funzione *sort,* in modo da poter quindi procedere alla scansione sequenziale accoppiata di af e tf; quindi la postcondizione del programma deve comprendere quella della funzione *sort* su tf, e cioè (α):

(tf') ordinato-su-tnum **and** (tf') esaurisce (tf)

– Su af: un qualunque conto corrente non viene modificato se su tf non sono presenti transazioni ad esso relative, mentre in caso contrario viene aggiornato con tutte le transazioni che lo riguardano. Formalmente si ha (β):


```
∀j in af'.dom: af'(j).anum ∈ af(j).anum and
   [(∀i in tf.dom: tf(i).tnum ∉ af(j).anum ⟹
      (af'(j).ainfo) coincide-con (af(j).ainfo)) and
    (∃i in tf.dom: tf(i).tnum ∈ af(j).anum ⟹
      ∀i in tf.dom (tf(i). tnum ∈ af(j).anum):
      (af(j).ainfo) aggiornamento con (tf(i).tinfo)
         in (af-(j).ainfo))]
```

dove il predicato "aggiornato-con" è lo stesso usato nella definizione della **ofun** update.

– Su rf: le specifiche del sistema non precisano se deve essere stampata una riga di resoconto per ogni numero di conto presente in af o solo per quelli su cui sono state effettuate transazioni: qui di seguito è fatta e formalizzata questa seconda scelta. Inoltre, anche se non esplicitamente richiesto dalle specifiche del sistema, una riga di resoconto contenente un messaggio di errore è prodotta per tutti quei numeri di conto che figurano nell'archivio delle transazioni senza che un corrispondente record sia presente nell'archivio dei conti correnti.

Detto TFinfo (n) l'insieme delle informazioni relative alle transazioni effettuate sul numero di conto n, cioè:

$$TFinfo(n) = \{ tf(i).tinfo \;/\; i \textbf{ in } tf.dom \textbf{ and } tf(i).tnum = n \}$$

queste considerazioni possono essere così formalizzate (γ):

```
∀K in rf'.dom: [(∃j in af.dom ∃i in tf.dom:
      [rf'(k).rnum ∈ af(j).anum ∈ tf(i).tnum ⟹
         ((rf'(k).rinfo) è-stand-msg and
         (rf'(k).rinfo) costruito-con
            (af(j).info,TFinfo (rf'(k).rnum))

   and
      (∃i in tf.dom: rf'(k).rnum ∈ tf(i).tnum and
      ∀j in af.dom: rf'(k).rnum ∉ af(j).anum) ⟹
         (rf'(k).rinfo è-error-msg)]
```

dove il predicato "costruito-con" è specificabile (cfr. clausola **Predicates**) utilizzando i predicati "creato-con" e "ottenuto-con" corrispondenti rispettivamente alle funzioni *create* e *aggiorna,* mentre il predicato "è-error-msg" è lo stesso usato nella postcondizione della funzione *error.*

La postcondizione del programma è quindi data dalla congiunzione di (α), (β) e (γ) ed è riportata in fig. 4 (la definizione dell'insieme TFinfo, introdotto per semplificare la postcondizione è riportata come commento subito sotto la parola chiave **Specifications** in quanto TFinfo viene utilizzato anche nel **loop invariariant**).

Per quanto concerne l'asserzione invariante dell'iterazione più esterna, essa deve indicare che la funzionalità complessiva del programma (espressa dalla post condizione) è stata realizzata fino al punto a cui è giunta la scansione sequenziale dei due archivi af e tf (quest'ultimo già ordinato).

Ad ogni ciclo dell'iterazione siano j ed i gli indici che rappresentano rispettivamente gli elementi di af e tf da esaminare, sia N_i l'insieme dei numeri di conto distinti presenti in tf fino ad i escluso. Supponendo che l'elaborazione di transazioni con uguale numero di conto sia demandata ad iterazioni più interne (cioè supponendo che, se un numero di conto è già stato elaborato, allora sono state elaborate tutte le transazioni che lo riguardano, e che se una transazione non ha corrispondente conto corrente allora sono state bypassate tutte le transazioni con numero di conto a lei uguale), l'invariante si ricava direttamente dalla postcondizione e risulta essere quello riportato in fig. 4 (anche in questo caso la definizione dell'insieme N_i introdotto per semplificare l'invariante è riportata come commento).

Passando alla costruzione del programma, si nota che l'invariante garantisce che gli elementi correnti di af e tf, rispettivamente af(j) e tf(i), non sono stati ancora esaminati. Si devono allora anzitutto bypassare senza modificare tutti quegli elementi da af(j) in avanti che hanno numero di conto minore di quello di tf(i) e su cui pertanto non sono state effettuate transazioni. La scansione di af è sospesa quando si incontra un suo elemento con numero di conto uguale o maggiore di tf(i).tnum o quando sono stati esaminati tutti gli elementi di af senza che sia stato trovato un elemento che soddisfi una di queste condizioni.

Prevedendo per questi tre casi un differente trattamento, il programma deve quindi avere come corpo dell'iterazione più esterna:

```
do  j ⩽ af.hib cand af(j).anum.ismin(tf(i).tnum)
      ⟶ j:=j+1
od;
if  j ⩽ af.hib cand af(j).anum.iseq(tf(i).tnum)
      ⟶ CASO 1
□  j ⩽ af. hib cand af(j).anum.ismaj(tf(i).tnum)
      ⟶ CASO 2
□  j > af. hib  ⟶  CASO 3
fi
```

dove **cand** è l'operatore **and** condizionale che garantisce che il secondo termine della condizione sia esaminato solo se il primo è soddisfatto; ciò è necessario perchè quando j > af.hib, af(j) è indefinito e quindi non gli si possono applicare le funzioni *iseq, ismin* e *ismaj.*

L'elaborazione da effettuare nei tre casi previsti è anch'essa ricavabile dall'invariante. Infatti:

CASO 1: per la i-esima transazione si trova un corrispondente elemento in af; questo elemento viene allora aggiornato (**ofun** update) e, utilizzando una variabile locale r di tipo report, viene creata una nuova riga di resoconto (**ofun** create); vengono quindi esaminate tutte le successive transazioni con uguale numero di conto utilizzando le relative informazioni in esse contenute per aggiornare il corrispondente elemento di rf (**ofun** update) e la riga di resoconto r (**ofun** modify), che viene alla fine accodata a rf utilizzando la funzione standard *hiext.*

Nel programma, quindi, il comando corrispondente alla condizione af(j).anum.iseq(tf(i).tnum) sarà:

```
r: create (af(j));
do i ⩽ tf.hib cand af(j).anum.iseq(tf(i).tnum) —>
        af(j): update (tf(i));
        v: modify (tf(i));
        i: = i+1
od;
rf: hiext(r); j: j+1
```

CASO 2: il primo elemento di af con numero di conto non minore di tf(i).tnum ha numero di conto maggiore di tf(i).tnum, cioè non esiste in af un conto corrente con numero uguale a quello di tf(i).
Si tratta di una situazione di errore che deve essere segnalata con una opportuna riga di resoconto (**ofun** error), mentre tutte le transazioni che eventualmente seguono tf(i) e hanno uguale numero di conto vanno bypassate.

Nel programma, quindi, il comando corrispondente alla condizione af(j).anum.ismaj(tf(i).tnum) sarà:

```
r:error (tf(i));
rf: hiext (r);
i: = i+1;
do i < tf.hib cand rf.high.rnum.iseq(tf(i).tnum) —>
    i: = i+1
od
```

CASO 3: tutti gli elementi di af vengono esaminati senza che ne sia trovato uno con numero di conto uguale o maggiore di tf(i).tnum. In questo caso, la segnalazione di errore (**ofun** error) è necessaria non solo per tf(i).tnum, ma anche per tutti i numeri di conto successivi di tf.

Nel programma quindi alla condizione j>af.hib corrisponderà il comando:

```
do i⩽tf.hib —>  r:error (tf(i));
                rf: hiext(r);
                i: i+1;
                do i < tf.hib cand
                   rf.high.rnum.iseq(tf(i).tnum)
                   —> i:=i+1
                od
od
```

Ricavata la struttura del corpo della iterazione (detto S) si tratta di definirne i limiti. Naturalmente af e tf vanno esaminati dall'inizio. Inoltre, poichè in S è già inserita una iterazione su af, delimitata su af.hib, per l'iterazione più esterna è sufficiente il controllo su tf. Pertanto si ha:

```
i:= tf.lob;
j:= af.lob;
do i⩽tf.hib —> S od
```

Se, quando l'iterazione su tf termina, af non è terminato ciò significa che esistono altri numeri di conto su cui non sono state effettuate transazioni; ma avendo scelto di non riportare sul resoconto i numeri di conto cui non corrisponde alcuna transazione, ciò non richiede nessuna ulteriore elaborazione.
Questo fatto può essere formalmente verificato controllando che, quando è soddisfatta la condizione di terminazione dell'iterazione più esterna (cioè per i>tf.hib, o più precisamente, i=tf.hib+1), l'invariante implica la postcondizione.

La garanzia che l'iterazione termini è data dalla possibilità di definire una funzione variante decrescente dell'indice i che la controlla. Essa è:

y= tf.hib - i

Poichè nella costruzione dell'invariante e del programma si è fatto uso dell'ipotesi di ordinamento di af e tf e poichè per quest'ultimo la precondizione non la garantisce, è necessario completare il programma con la preliminare esecuzione della funzione *sort* su tf, prodotta nel comando tf:sort.

Infine, è necessario completare le inizializzazioni, definendo quale è il valore iniziale dell'indice intero che scorre su rf. Nel caso in esame una qualunque scelta è possibile, in quanto il programma si limita ad accodare a rf nuovi elementi. Sarà pertanto per semplicità:

rf:= (1)

La documentazione completa del programma, è data in fig. 4.

Per completare la documentazione del disegno della prima macchina si devono specificare, nella clausola **Predicates,** tutti i predicati utilizzati nella definizione delle funzioni e dei programmi. Tale formalizzazione è riportata in fig. 4. In essa molti predicati astratti non sono stati specificati, utilizzando un testo in linguaggio naturale in quanto di immediata comprensione. Per lo stesso motivo, per il secondo gruppo, non sono state formalizzate le proprietà.
Invece, ove possibile, è stata data la specificazione di predicati più generali (ad esempio quelli di ordinamento degli archivi) in termini di predicati più elementari (quelli sulle relazioni d'ordine sui singoli elementi).

3.3 Il secondo livello del disegno del sistema

Nella macchina di primo livello M1 quattro sono i tipi definiti **abstract,** che devono quindi essere raffinati: *aaltro, taltro, raltro* e *num.* È abbastanza evidente, ad esempio considerando i predicati su di essi definiti, la connessione che esiste tra i primi tre, che spinge a specificarli in un'unica macchina (M-altro).
D'altra parte, però, il loro raffinamento non è indipendente da quello del tipo *num* (che si può ipotizzare fatto

in un'altra macchina, M-num): infatti nelle pre e post condizioni delle funzioni definite in M1 compaiono predicati che mettono in relazione oggetti di tipo num con oggetti di tipo *aaltro, taltro, raltro.*

Queste considerazioni evidenziano tre possibili scelte per lo sviluppo del sistema, ciascuna caratterizzata da una diversa gerarchia delle macchine:

a) raffinare i quattro tipi abstract di M1 in un'unica macchina di secondo livello (fig. 5 /a)

b) definire due macchine M-num e M-altro di ugual livello gerarchico rispetto a M1 ma non indipendenti tra loro (fig. 5 /b)

c) definire due macchine M-num e M-altro, l'una gerarchicamente subordinata all'altra (fig. 5 /c)

Nel seguito si è scelta la seconda alternativa che ha, rispetto alla prima, il vantaggio di consentire una scomposizione del secondo livello di disegno del sistema, aumentandone la possibilità di controllo e comprensione senza incrementarne eccessivamente la complessità; rispetto alla terza, il vantaggio di non introdurre una gerarchia tra le macchine M-num e M-altro che risulterebbe nel complesso immotivata.

La macchina M-num realizza un generico oggetto di tipo *num,* specificandone una rappresentazione che consenta di scrivere i programmi che realizzano le funzioni definite in M1 sul tipo *num* (*iseq, ismin, ismaj*) e la funzione *sort* definita in M1 sul tipo *transaction-file.*

La macchina M-altro, rappresentando un generico oggetto di tipo *aaltro,* uno di tipo *taltro* ed uno di tipo *raltro*, consente di scrivere i programmi che realizzano le funzioni *update, create, modify, error* definite nella macchina di primo livello sui tipi *account* e *report.*

La scelta fatta però, come evidenziato dalla figura, ha la conseguenza di indurre un intreccio nel disegno delle due macchine; in altre parole sviluppando M-num si individua la necessità di "arricchire" M-altro e viceversa.
Si tenga presente che, a causa di ciò, la documentazione completa delle due macchine (mostrata rispettivamente in fig. 6 e in fig. 7, a cui nel testo talvolta si rimanda) può contenere particolari non ancora presentati.

3.4 La macchina M-num

Nella definizione di un nuovo tipo atto a rappresentare un numero di conto, notiamo che le specifiche del problema non ne precisano la struttura.
Si è ritenuto verosimile individuare un numero di conto con una coppia (x,y), dove x indica una priorità individuata dalle prime tre lettere dell'alfabeto, e y è un'intero positivo al massimo di cinque cifre. Formalmente:

```
MACHINE M-num
Refines num
Types
number: (prec.(A □ B □ C); card: integer)
  let h: number
  inv 0‹ h.card ‹ 99999
end number
```

Nella clausola **Permanent data** devono essere dichiarati gli oggetti su cui la macchina M-num opera. Poichè questa deve definire, esplicitandola nella clausola **Mapping,** una rappresentazione per le variabili di tipo *num* utilizzate nella prima macchina e per i predicati su di esse definiti, è necessario introdurre due variabili di tipo *number* in modo da rendere possibile la specificazione dei predicati binari ⊜ e ⓒ .
Quindi sarà:

```
Permanent data
  m,n : number
```

Il fatto che m ed n rappresentino due generici oggetti di tipo *num,* come definito nella macchina di primo livello, è così formalizzato:

```
Mapping
  let a,b:num
  m represents a
  n represents b
```

La clausola **Mapping** deve essere completata specificando come i predicati della macchina gerarchicamente inferiore (M-num) rappresentano quelli della macchina gerarchicamente superiore garantendo che ne siano soddisfatte le proprietà caratteristiche. In altre parole, è necessario "tradurre" i predicati astratti definiti nella macchina di primo livello su *num* nei termini "adatti" alla macchina M-num, cioè come predicati sul tipo *number.*
Si tratta, perciò, di definire sugli oggetti di tipo *number* una relazione d'ordine ed una relazione di uguaglianza capaci di rappresentare le corrispondenti relazioni definite nella macchina di primo livello e caratterizzate dalla proprietà p1,...,p7 (cfr. fig. 4)

I predicati su M-num (che devono in particolare esprimere la relazione d'ordine tra le priorità: priorità A maggiore priorità B maggiore priorità C) sono così definiti, e la clausola **Mapping** così completata:

```
Predicates
  let x,y: number;
    (x) EQ (y) means (x.prec=y.prec and
                      x.card=y.card)
    (x) MIN (y) means ((x.prec=y.prec and
                        x.card⩽y.card)or
                       (x.prec=B and y.prec=A) or
                       (x.prec=C and not y.prec=C))
    (x)MAJ(y) means (not (x)EQ(y) and
                     not (x)MIN(y))
```

Fig. 5/a

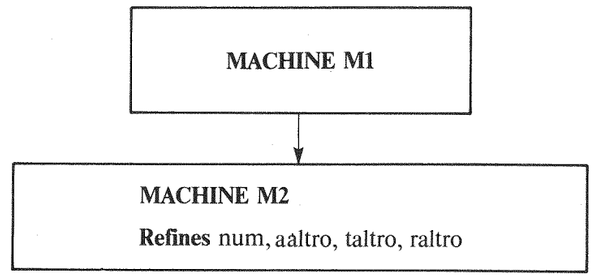


Fig. 5/b

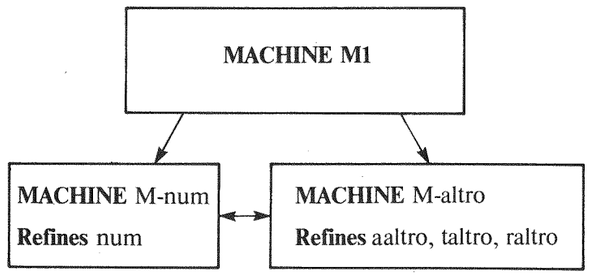


Fig. 5/c

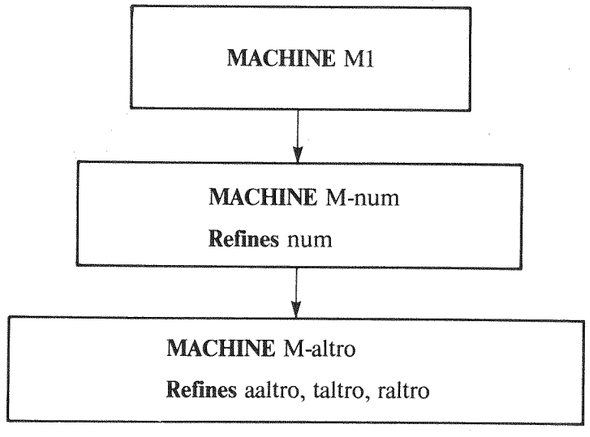


Fig. 5: Le tre possibili scelte per il secondo livello gerarchico del disegno del sistema

MACHINE M-num

MISSION DESCRIPTION

. . . . . .

DESIGN DOCUMENTATION

Refines num, transaction-file, transaction, taltro

Predicates

let x,y: number

(x) EQ (y) **means** (x.prec=y.prec **and** x.card=y.card)

(x)MIN(y) **means** ((x.prec=y.prec **and** x.card ‹ y.card)**or**
(x.prec=B **and** y.prec=A) **or**
(x.prec=C **and not** y.prec=C))

(x)MAJ(y) **means** (**not** (x)EQ(y) **and not** (x)MIN(y))

let A,B: transaction

(A.tinfo) uguale-a (B.tinfo) **means abstract**

Types

number: (prec: (A □ B □ C); card: integer)

let h: number

inv 0 ‹ h. card ‹ 99999

end number

transaction-file: transaction **array**

transaction: (tnum: number; tinfo: taltro)

let x: transaction

ofun copy (y: transaction)

pre: (y.tnum) EQ (x.tnum)

post: (x'.tinfo) uguale-a (y.tinfo)

end transaction

taltro: **abstract**

Permanent data

m,n: number

t: transaction-file

Mapping

let a,b: num

(m) **represents** (a)

(n) **represents** (b)

(m) EQ (n) **represents** a ⊜ b

(m) MIN (n) **represents** a ⟨ b

(m) MAJ (n) **represents** a ⟩ b

PROGRAMS

iseq

ismin

ismaj

sort

assign

PROGRAM iseq (p: number) **returns** (b: boolean)

precondition true

postcondition b=true ⟺ (n) EQ (p)

P-notation

if (n.prec = p.prec **and** n.card = p.card) ⟶ b:=true

□ (n.prec ≠ p.prec **or** n.card ≠ p.card) ⟶ b:=false

fi

end iseq

PROGRAM ismin (p: number) **returns** (b: boolean)

precondition true

postcondition b=true ⟺ (n) MIN (p)

P-notation

if n.prec = p.prec ⟹ **if** n.card ‹ p.card ⟶ b:=true
□ n.card ⩾ p.card ⟶ b:=false
fi

□ p.prec = A **and not** (n.prec = A) ⟶ b:=true

□ p.prec = B **and** n.prec = A ⟶ b:=false

□ p.prec = B **and** n.prec = C ⟶ b:=true

□ p.prec = C **and not** (n.prec=C) ⟶ b:=false

fi

end ismin

PROGRAM ismaj (p: number) **returns** (b: boolean)

precondition true

postcondition b=true ⟺ (n) MAJ (p)

P-notation

if n.prec = p.prec ⟹ **if** n.card › p.card ⟶ b:=true
□ n.card ⩽ p.card ⟶ b:=false
fi

□ n.prec = A **and not** (p.prec = A) ⟶ b:=true

□ n.prec = B **and** p.prec = A ⟶ b:=false

□ n.prec = B **and** p.prec = C ⟶ b:=true

□ n.prec = C **and not** (p.prec = C) ⟶ b:=false

fi

end ismaj

PROGRAM assign (p: number)

precondition true

postcondition (n) EQ (p)

P-notation

n.prec := p.prec ;

n.card := p.card

end assign

PROGRAM sort

precondition true

postcondition t.lob ‹ i ‹ t.hib ⟺ [(t(i).tnum)MIN(t(i+1).tnum) **or**
(t(i).tnum) EQ (t(i+1).tnum))]
and
[∀i **in** t'.dom ∃j **in** t.dom :
((t'(i).tnum) EQ (t(j).tnum) **and**
(t'(i).tinfo)uguale-a(t(j).tinfo))]

P-notation

. . . . . .

end sort

Fig. 6: La macchina M-num

Mapping
let a,b: num
(m) **represents** (a)
(n) **represents** (b)
(m) EQ (n) **represents** a ⊜ b
(m) MIN (n) **represents** a ⓘ b
(m) MAJ (n) **represents** a ⓙ b

È immediato verificare che la rappresentazione qui definita garantisce che sono verificati l'invariante sul tipo *num* e le proprietà p1,...,p7 sui predicati ⊜ e ⓘ definiti in M1. Si noti che, in M1, sul tipo *num* erano definiti anche i predicati ⓚ e ⋐ di cui nel mapping non si è esplicitata una rappresentazione.
Questa è comunque già definita a partire da quella di ⊜ e ⓘ e dalla definizione di ⓚ e ⋐ in funzione di questi (cfr. fig. 4). Anche per ⓙ varrebbero analoghe considerazioni, ma si è invece preferito esplicitarlo poichè al contrario di ⋐ e ⓚ compare nella postcondizione della funzione *ismaj*.

Le pre e post condizioni delle tre funzioni definite nella prima macchina sul tipo *num*, possono a questo punto essere espresse nei termini della macchina M-num e le funzioni possono venire realizzate da semplici programmi.

Program iseq (p: number) **returns** (b: boolean)
precondition true
postcondition b=true <⟹> (n) EQ (p)
text
if (n.prec=p.prec **and** n.card=p.card) ⟶ b:=true
□ (n.prec=p.prec **or** n.card≠p.card) ⟶ b:=false
fi
end iseq

Program ismin (p:number) **returns** (b:boolean)
precondition true
postcondition b=true <⟹> (n) MIN (p)
text
if n. prec=p.prec ⟶ **if** n.card‹p.card ⟶ b:=true
□ n.card ≥ p.card ⟶ b:=false
fi
□ p.prec=A **and not** (n.prec=A) ⟶ b:=true
□ p.prec=B **and** n.prec=A ⟶ b:=false
□ p.prec=B **and** n.prec=C ⟶ b:=true
□ p.prec=C **and not** (n.prec=C) ⟶ b:=false
fi
end ismin

Program ismaj (p:number) **returns** (b:boolean)
precondition true
postcondition b=true <⟹> (n) MAJ (p)
text
if n.prec=p.prec ⟹ **if** n.card › p.card ⟶ b:=true
□ n.card≤p.card ⟶ b:=false
fi
□ n.prec=A **and not** (p.prec=A) ⟶ b:=true
□ n.prec=B **and** p.prec=A ⟶ b:=false
□ n.prec=B **and** p.prec=C ⟶ b:=true
□ n.prec=C **and not** (p.prec=C) ⟶ b:=false
fi
end ismaj

L'avere definito una rappresentazione per il tipo *num* consente anche di specificare come programma la funzione *sort* sul tipo *transaction-file*.
Per farlo è però necessario definire in M-num un oggetto di un tipo corrispondente al tipo *transaction-file* di M1.

La clausola **Types** di M-num va quindi ampliata a contenere il corrispondente del tipo *transaction-file*, e necessariamente anche di *transaction-* e *taltro*.
I primi due mantengono esattamente la stessa struttura che era già stata definita in M1; il terzo rimane **abstract**, e su di esso deve essere definita una funzione *copy* che sarà esplicitata come programma (sequenza di assegnamenti) nella macchina M-altro in cui è definita una rappresentazione più dettagliata per la seconda componente di una generica transazione.

Tutte le volte che la struttura di un tipo non viene raffinata (o perchè già specificata o perchè rimane **abstract** ed al più nuove funzioni vengono introdotte su di esso) conveniamo, per non sovraccaricare la documentazione, di utilizzare gli stessi identificatori per i tipi, le loro componenti, i relativi predicati in modo da non dover esplicitare le corrispondenze. L'insieme dei tipi a cui si applica questa convenzione è facilmente individuabile nella documentazione poichè essi sono elencati nella clausola **Refines** ma non figurano nella clausola **Mapping.**

Per la macchina M-num si giunge così alla forma completa presentata in fig. 6.
Le pre e post condizioni del programma *sort* sono ottenute da quelle definite per la funzione *sort* in M1, sostituendo i predicati ⊜, ⓘ, ⓚ con i corrispondenti EQ, MIN e loro combinazioni, mentre, secondo quanto convenuto prima, il predicato astratto "uguale-a" sul tipo *transaction* non viene modificato (ne sarà data una rappresentazione in M-altro).
Il testo del programma *sort* non è esplicitato in quanto si può ricorrere ad un qualunque algoritmo di sort per i files ed in particolare al sort messo a disposizione dal sistema; infatti la sua applicazione ne soddisfa la postcondizione.

3.5 La macchina M-altro

Le informazioni associate ad un dato numero di conto che, insieme a questo, costituiscono il generico elemento degli archivi di conti correnti, transazioni e resoconti, lasciate astratte nella macchina di primo livello, sono ricavabili dai requisiti del sistema. Esse riguardano:
- somme di denaro, che vengono rappresentate da variabili di tipo *real;*
- l'indicazione se per una data operazione bancaria (indicata in tf) si tratta di un deposito (rappresentato dal carattere D) o di un prelievo (carattere P);
- il numero di operazioni bancarie eseguite (numero complessivo dei depositi e dei prelievi effettuati su un dato conto), da indicare nel resoconto, specificato con variabili di tipo *integer.*

La definizione dei tipi di M-altro che devono raffinare i tipi *aaltro, taltro, raltro* definiti in M1 risulta pertanto essere:

MACHINE M-altro
Refines aaltro, taltro, raltro
Types
denaro: real
aother: denaro
tother: (oper:(D □ P); lit:denaro)
rother: (stand-msg:standard-message □
err-msg: error-message)
standard-message: (saldoin:denaro; saldofin:denaro;
ndep:integer; nprel:integer)
error-message: character **array**

dove *aother* indica il saldo per il numero di conto corrispondente; *tother* indica la somma di denaro prelevata o depositata in una transazione sul numero di conto corrispondente; *rother* indica che ogni riga di resoconto o specifica, per il numero di conto corrispondente, il saldo prima e dopo le transazioni ad esso relative affiancati dal numero complessivo di depositi e di prelievi, oppure è costituita da un messaggio di errore non ulteriormente strutturato (si noti che questa possibilità di un doppio tipo di resoconto era "annunciata" nella macchina M1 dalla definizione sul tipo report dei due predicati "è-stand-msg" ed "è-error-msg").
Queste considerazioni sono frutto di una *corrispondenza intuitiva* che il progettista ha già in mente quando definisce dei predicati su tipi astratti in una macchina (nel nostro caso sui campi *ainfo, tinfo* e *rinfo* nella macchina di primo livello) destinati a essere raffinati in macchine di livello inferiore (M-altro). È proprio tale corrispondenza intuitiva che viene poi formalizzata nell'apposita sezione **Mapping**, dopo aver definito nella clausola **Predicates** (cfr. fig. 7) sui tipi della macchina M-altro dei predicati che costituiscono la rappresentazione di quelli definiti in M1.
Introducendo un opportuno insieme di variabili permanenti in M-altro è possibile definire la rappresentazione per oggetti e predicati definiti in M1 sui tipi *aaltro, taltro, raltro* in modo che ne siano soddisfatte le proprietà.
Da questo punto di vista sarebbe sufficiente introdurre variabili di tipo *aother, tother, rother*. Poichè però, nello specificare come programmi le funzioni *update, create, modify, error* definite in M1 sui tipi *account* e *report* si incorre nella necessità di avere a disposizione variabili di tipo *account, report* e *transaction* questi tipi devono essere disponibili in M-altro e vanno quindi inseriti nelle clausole **Refines** e **Types** (assumendo anche in questo caso la convenzione sul mapping banale) e tra i **Permanent data** risulta più opportuno dichiarare addirittura le variabili corrispondenti (cfr. fig. 7).

Si può ora passare alla realizzazione delle funzioni *update, create, modify, error* come programmi. Nel caso della seconda e della quarta, che hanno nella postcondizione il predicato astratto ⊜ sulla componente di tipo *num*, per farlo è necessario definire sul tipo *num* una funzione di assegnamento che sarà specificata come sequenza di assegnamenti nella macchina M-num in cui è definita la rappresentazione del tipo *num.*
Le clausole **Refines** e **Types** di M-altro vanno quindi integrate come mostrato in fig. 7, dove sono pure presentate le **Specification** e la **P-notation** dei quattro programmi.

Per ultimare il disegno delle due macchine di secondo livello è necessario specificare come programmi:
- in M-num, la funzione *assign* introdotta in M-altro sul tipo *num*
- in M-altro, la funzione *copy* introdotta in M-num sul tipo *transaction.*

Si noti che, grazie alla convenzione addottata di mantenimento dei nomi su cui si effettua il mapping banale non è necessario alcun ulteriore ampliamento delle corrispondenti clausole delle due macchine, oltre a quello qui sotto indicato.

MACHINE M-num
Permanent data
n: number
Program assign (p:number)
precondition true
postcondition (n) EQ (p)
P-notation
n.prec := p.prec ;
n.card := p.card
end assign

MACHINE M-altro
Permanent data
t: transaction
Program copy (y: transaction)
precondition y.tnum ⊜ t.tnum
postcondition (t'.tinfo) equal-to (y.tinfo)
P-notation
t.oper := y.oper ;
t.lit := y.lit
end copy

3.6 Dal disegno alla programmazione

Se è vero che i vincoli imposti al sistema dall'ambiente in cui esso deve operare (vincoli hardware - le risorse disponibili - e software - il sistema operativo ed il linguaggio di programmazione scelto -) possono condizionare il disegno anche ai primi livelli gerarchici (ad esempio se è noto dai requisiti che si avranno a disposizione come strutture di dati dinamiche solo files sequenziali è inopportuno progettare su array che con essi dovranno essere rappresentati delle funzioni che implicano la necessità di un accesso diretto) è tuttavia

```
MACHINE M-altro
MISSION DESCRIPTION
. . . . . .
DESIGN DOCUMENTATION
  Refines aaltro, taltro, raltro, transaction, account, report, num
  Predicates
    let A , A' : aother
        B , B' : tother
        C , C' : rother
        (B') equal-to (B) means [B'.oper = B.oper and B'.lit = B.lit]
        (A) updated-by (B) means [(B.oper = D ⟹ A' = A + B.lit) and
                                  (B.oper = P ⟹ A' = A-B.lit) ]
        (C) created-by (A) means  [C.tag = stand-msg ⟹ C.saldoin = A and C.saldofin = A and C.ndep = Ø   and C.nprel = Ø   ]
        (C) modified-by (B) in (C) means [(C.tag. = stand.msg and B.oper = D ⟹
                                          C'.saldoin = C.saldoin and C'.saldofin=C.saldofin + B.lit and
                                          C'.ndep=C.ndep + 1 and C'.nprel=C.nprel)
                                      and  (C.tag=stand-msg and B.oper=P ⟹
                                          C'.saldoin=C.saldoin and C'.saldofin=C.saldofin-B.lit  and
                                          C'.ndep=C.ndep and C'.nprel=C.nprel + 1)]
        (C) is-stand-msg means [C.tag=stand-msg]
        (C) is-error-msg means [C.tag=err-msg]

  Types
    denaro: real
    aother: denaro
    tother: (oper: (D□P); lit: denaro)
    rother: (stand-msg: standard-message□err-msg: error-message)
    standard-message: (saldoin: denaro; saldofin: denaro; ndep: integer; nprel: integer)
    error-message: character array
    num: abstract
      let x: num
      ofun assign (y: num)
        pre  true
        post x' = y
    end num
    transaction: (tnum: num; tinfo: tother)
    account: (anum: num; ainfo: aother)
    report: (rnum: num; rinfo: rother)
  Permanent data
    a , b : account
    t , s : transaction
    r , v : report
  Mapping
    let x,x' : aaltro
        y,y' : taltro
        z,z' : raltro
    a.ainfo represents x
    b.ainfo represents x'
    t.tinfo represents y
    s.tinfo represents y'
    r.rinfo represents z
    v.rinfo represents z'
    b.ainfo = a.ainfo represents (x') coincide-con (x)
    (s.tinfo)equal-to(t.tinfo) represents (y')uguale-a(y)
    (a.ainfo)updated-by(t.tinfo) in (b.ainfo) represents (x)aggiornato-con(y) in (x')
    (r.rinfo)created-by(a.ainfo) represents (z)creato-con(x)
    (r.rinfo)modified-by(t.tinfo) in (v.rinfo) represents (z)modificato-con(y) in (z')
    (r.rinfo)is-stand-msg represents (z)è-stand-msg
    (r.rinfo)is-error-msg represents (z)è-error-msg
```

Fig. 7: La macchina M-altro


```
PROGRAMS
      update
      create
      modify
      error
      copy

PROGRAM update (y: transaction)
  precondition true
  postcondition (a.ainfo) update-by (y.tinfo) in (a'.ainfo)
  P-notation
    if y.tinfo.oper = D ⟶  a.ainfo:=a.ainfo + y.tinfo.lit
    □ y.tinfo.oper = P ⟶  a.ainfo:=a.ainfo - y.tinfo.lit
    fi
end update

PROGRAM create(x: account)
  precondition true
  postcondition r.rnum = x.anum and (r.rinfo)is-stand-msg
                and (r.rinfo)created-by(a.ainfo)
  P-notation
    r.rnum: assign (x.anum);
    r.rinfo.saldoin:=x.ainfo;
    r.rinfo.saldoin:=x.ainfo;
    r.rinfo.ndep:=Ø;
    r.rinfo.nprel:=Ø
end create

PROGRAM modify (y: transaction)
  precondition r.rnum = y.tnum and (r.rinfo)is-stand-msg
  postcondition (r.rinfo)modified-by(y.tinfo) in (r'.rinfo)
  P-notation
    if y.tinfo.other = D ⟶
                    r.rinfo.saldofin:=r.rinfo.saldofin + y.tinfo.lit;
                    r.rinfo.ndep:=r.rinfo.ndep + 1
    □ y.tinfo.oper = P ⟶
                    r.rinfo.saldofin:=r.rinfo.saldofin - y.tinfo.lit;
                    r.rinfo.nprel:=r.rinfo.nprel + 1
    fi
end modify

PROGRAM error (y: transaction)
  precondition true
  postcondition r.rnum = y.tnum and (r.rinfo)is-error-msg
  P-notation
    r.rnum : assign (y.tnum) ;
    r.rinfo :=  'il numero di conto indicato nella transazione
                non ha corrispondente nell'archivio conti correnti'
end error

PROGRAM copy (y: transaction)
  precondition y.tnum = t.tnum
  postcondition (t'.tinfo) equal-to (y.tinfo)
  P-notation
    t.oper:=y.oper
    t.lit:=y.lit
end copy
```

Fig. 7 (continuazione)

opportuno che le considerazioni di implementazione intervengano a condizionare il disegno il piú tardi possibile onde garantire la sua massima indipendenza da eventuali cambiamenti dei vincoli implementativi.
Gli ultimi livelli del disegno sono invece generalmente condizionati in modo rilevante dai vincoli hardware e software imposti dall'ambiente al sistema.
Se ad esempio in una macchina è definito il tipo astratto "albero" caratterizzato dalle usuali funzioni per aggiungere o togliere sottoalberi ad un nodo, la sua rappresentazione è differente a seconda che si sia scelto come linguaggio di programmazione il Pascal (che consente di rappresentare gli alberi con i puntatori) o il Fortran (che induce necessariamente la rappresentazione vettoriale).
Dal punto di vista delle modalità del raffinamento dei dati, questa differenza tra i primi e gli ultimi livelli del raffinamento si traduce in un maggior utilizzo, rispettivamente, della "data expansion" e della "data representation" (cfr[1], par. 14 e 15).
Benchè il confine tra disegno e implementazione non sia univocamente identificabile, il disegno del sistema può generalmente dirsi concluso quando il raffinamento delle strutture di dati garantisce che per tutti i tipi è definita una rappresentazione in termini immediatamente traducibili nell'ambiente di calcolo prescelto.
Nell'esempio qui considerato il livello di raffinamento del disegno raggiunto è adeguato ai vincoli imposti dalla scelta del Pascal come linguaggio di programmazione e non è quindi necessario definire nessuna ulteriore macchina. Infatti il programma, che realizza il sistema disegnato per rispondere ai requisiti fissati, è ottenuto dalla documentazione di disegno sulla base delle seguenti considerazioni:

a) *sui tipi di dati* valgono le seguenti corrispondenze (equivalenti ad un "mapping" tra tipi definiti nel programma Pascal e tipi definiti nelle macchine M1, M-num e M-altro);

- la struttura di dati **file** del Pascal consente di rappresentare gli oggetti af, tf, rf definiti in M1, ma è necessario introdurre un ulteriore oggetto, corrispondente di af', detto afnew, in quanto il Pascal standard non consente di effettuare operazioni di aggiornamento (richieste dall'uso della **ofun** update) su di un file che è scandito sequenzialmente in lettura (l'incremento dell'indice j che scorre su af è tradotto in Pascal con una get (af));
- la struttura di dati **record** del Pascal consente di rappresentare tutti i tipi definiti con l'operatore di concatenazione e l'opzione **variant** consente di rappresentare il tipo *rother* definito in M-altro come alternation;
- i tipi **standard** di Wellmade *integer, real boolean* e *char* hanno un immediato corrispondente in Pascal;
- i due tipi definiti per **enumerazione**
  prec: (A □ B □ C) introdotto in M-num e
  oper: (D □ P) introdotto in M-altro

sono rispettivamente rappresentati in Pascal come:

prec: "A" . . "C"
oper: (DEP,PREL)

Tali definizioni inducono una relazione d'ordine sui due tipi *prec* e *oper* data rispettivamente da: "A" ‹ "B" ‹ "C" e DEP ‹ PREL.
La seconda è superflua; la prima invece contraddice la relazione di ordinamento definita dal predicato MIN della macchina M-num su oggetti di tipo number, in cui si formalizza per la componente *prec* la relazione "A" › "B" › "C".
È pertanto necessario non utilizzare nel programma Pascal gli operatori ‹ e › applicati a variabili di tipo *prec*, ma esprimere la corretta relazione d'ordine utilizzando l'operatore = e la sua negazione.

b) *per quanto riguarda il controllo* è in generale necessario sciogliere il non determinismo insito nel linguaggio dei Guarded Commands. Nel caso in esame però nessuno dei programmi presenta un effettivo non determinismo, in quanto le condizioni inserite nei costrutti interattivi e selettivi sono sempre mutuamente disgiunte. È pertanto sufficiente tradurre ogni **do**...**od** nella più opportuna struttura interattiva del Pascal (in questo caso viene scelto sempre il **while**...**do**...) e ogni **if**...**fi** in un opportuno annidamento di **if**...**then**...**else**... (non essendoci mai le condizioni per utilizzare il **case**).
Ogni espressione logica che controlla l'esecuzione di un'istruzione di controllo va anch'essa opportunamente tradotta introducendo, in luogo della funzione **hib** sugli array, la funzione **eof** sui files. Per esprimere l'operatore logico **cand** (**and** condizionale), introdotto in M1 per evitare che le funzioni *iseq, ismin, ismaj* sul tipo *num* siano applicate a parametri indefiniti, è necessario introdurre una variabile booleana ausiliaria ed una struttura selettiva innestata in quella la cui condizione è espressa utilizzando il **cand.** Più precisamente, se nel programma di M1 compare una struttura del tipo:

do k ‹ f.hib **cand** f(k).F(g(h)) —> S **od**

dove f e g sono due array e k ed h sono gli indici che scorrono su di essi, F è una delle funzioni *iseq, ismin, ismaj* ed S l'insieme dei comandi da eseguire quando la condizione è soddisfatta, essa verrà tradotta in Pascal dalla sequenza di istruzioni

```
q:=true;
while not eof(f) and q do
   if F(f(k),g(h)) then S
                   else q:=false
```

dove q è una variabile booleana.

c) *la struttura complessiva del programma Pascal* è quella del programma della macchina di primo livello in cui:

- **ofun** e **vfun** sono realizzate come procedure e funzioni. Si noti che in questo passaggio dal linguaggio di disegno al Pascal è necessario che l'oggetto a cui la


```
(C) 1978 HONEYWELL GCOS LEVEL 62 PASCAL SOURCE LISTING 81/06/12 15:23:39 RU6.00

0048 --    PROGRAM CURRENTACCOUNTUPDATING (AF,TF,AFNEW,RF,OUTPUT);
0048 --    (*$U+*)
0048 --      CONST ERRORMSG =
0048 --            'LA TRANSAZIONE CON NUMERO INDICATO NON HA CONTO CORRISPONDENTE';
0048 --            INTESTAZIONETABELLA =
0048 --       NUMERO CONTO *  SALDO INIZIALE  *  SALDO FINALE  *  DEPOSITI  *  PRELIEVI ';
0048 --      TYPE NUM = RECORD PREC: 'A'..'C' ;
0048 --                        CARD: 1..MAXINT
0048 --                  END;
0048 --           ACCOUNT = RECORD ANUM:NUM;
0048 --                            AINFO:REAL
0048 --                    END;
0048 --           TRANSACTION = RECORD TNUM:NUM;
0048 --                                OPER:(DEP,PREL);
0048 --                                LIT: REAL
0048 --                              END;
0048 --           RINFO=(STANDARD,ERRORE);
0048 --           REPORT = RECORD RNUM:NUM;
0048 --                CASE RI: RINFO OF
0048 --                           STANDARD:(SALDOIN:REAL;
0048 --                                     SALDOFIN:REAL;
0048 --                                     NDEP:INTEGER;
0048 --                                     NPREL:INTEGER);
0048 --                           ERRORE:( )
0048 --                       END;
0048 --           ACCOUNTFILE = FILE OF ACCOUNT;
0048 --           TRANSFILE = FILE OF TRANSACTION;
0048 --           REPORTFILE = FILE OF REPORT;
0048 --      VAR AF, AFNEW: ACCOUNTFILE;
0084 --          TF:TRANSFILE;
00AC --          RF:REPORTFILE;
00E4 --          Q: BOOLEAN;
00E8 -- A  FUNCTION ISEQ(N,P:NUM):BOOLEAN;
0000 0- A    BEGIN
00UA --        IF (N.PREC=P.PREC) AND (N.CARD=P.CARD)
0052 --           THEN ISEQ:=TRUE
0038 --           ELSE ISEQ:=FALSE
0042 -U A    END;
004C -- A  FUNCTION ISMIN(N,P:NUM):BOOLEAN;
0000 0- A    BEGIN
00UA --        IF N.PREC=P.PREC
00UA 1-           THEN  BEGIN
0016 --                    IF N.CARD<P.CARD THEN ISMIN:=TRUE
0022 --                                     ELSE ISMIN:=FALSE
002C -1                END
0032 1-           ELSE CASE P.PREC OF
003E --             'A' : ISMIN:=TRUE;
0048 --             'B' : IF N.PREC='A'  THEN ISMIN:=FALSE ELSE ISMIN:=TRUE;
0068 --             'C' : ISMIN:=FALSE
0068 -1             END
00A2 -U A    END;
00B4 -- A  PROCEDURE UPDATE(VAR X:ACCOUNT;Y:TRANSACTION) ;
0000 0- A    BEGIN IF Y.OPER=DEP
00UA --               THEN X.AINFO:=X.AINFO + Y.LIT
0014 --               ELSE X.AINFO:=X.AINFO - Y.LIT
002C -U A    END;
0044 -- A  PROCEDURE CREATE (VAR X:REPORT;Y:ACCOUNT);
0000 0- A    BEGIN  X.RNUM.PREC:=Y.ANUM.PREC;
002C --             X.RNUM.CARD:=Y.ANUM.CARD;
004E --             X.RI:=STANDARD;
0058 --             X.SALDOIN:=Y.AINFO;
0064 --             X.SALDOFIN:=Y.AINFO;
0070 --             X.NDEP:=0;
007A --             X.NPREL:=0
007A -U A    END;
009C -- A  PROCEDURE MODIFY (VAR X:REPORT; Y:TRANSACTION);
0000 0- A  BEGIN
00UA 1-      IF Y.OPER=DEP THEN  BEGIN
0014 --                            X.SALDOFIN:=X.SALDOFIN + Y.LIT;
0028 -1                            X.NDEP:=X.NDEP + 1   END
003A 1-                    ELSE  BEGIN
003E --                            X.SALDOFIN:=X.SALDOFIN - Y.LIT;
0052 -1                            X.NPREL:=X.NPREL + 1  END
0064 -U A  END;
006C -- A  PROCEDURE  ERROR(VAR X:REPORT;Y:TRANSACTION);
0000 0- A    BEGIN  X.RNUM.PREC:=Y.TNUM.PREC;
002C --             X.RNUM.CARD:=Y.TNUM.CARD;
004E --             X.RI:=ERRORE
004E -U A    END;
0000 0-    BEGIN
006E --    RESET(AF);
007A --    RESET(TF);
0086 --    REWRITE(RF);
0092 --    REWRITE(AFNEW);
```

Fig. 8: Il programma codificato in PASCAL su Honeywell Level 62

```
U09E --     WHILE NOT EOF(TF) DO
U0AE 1-       BEGIN
U0AE --       Q:=TRUE;
U0B4 --       WHILE NOT EOF(AF) AND Q DO
U0C8 2-         IF ISMIN(AF@.ANUM,TF@.TNUM) THEN BEGIN
U0E4 --                                          AFNEW@:=AF@;
U0EC --                                          PUT(AFNEW);
U0F8 --                                          GET(AF)
U104 -2                                          END
U104 --         ELSE Q:=FALSE;
U112 --       IF EOF(AF) THEN  WHILE NOT EOF(TF)  DO
U12E 2-                          BEGIN
U12E --                          ERROR(RF@,TF@);
U142 --                          PUT(RF);
U14E --                          GET(TF);
U15A --                          Q:=TRUE;
U16U --                          WHILE NOT EOF(TF) AND Q DO
U174 --                            IF ISEQ(TF@.TNUM,RF@.RNUM) THEN GET(TF)
U19C --                                                       ELSE Q:=FALSE;
U1AA -2                          END
U1AA --       ELSE IF ISEQ(AF@.ANUM,TF@.TNUM) THEN
U1CE 2-                          BEGIN
U1CE --                          CREATE(RF@,AF@);
U1E2 --                          AFNEW@:=AF@;
U1EA --                          Q:=TRUE;
U1FU --                          WHILE NOT EOF(TF) AND Q DO
U204 3-                            IF ISEQ(AF@.ANUM,TF@.TNUM) THEN BEGIN
U22U --                                                            UPDATE(AFNEW@,TF@);
U254 --                                                            MODIFY(RF@,TF@);
U248 --                                                            GET(TF)
U254 -3                                                            END
U254 --                                                       ELSE Q:=FALSE;
U262 --                          PUT(RF);
U26E --                          PUT(AFNEW);
U27A --                          GET(AF);
U286 -2                          END
U286 --       ELSE (*ISMAJ(AF@.ANUM,TF@.TNUM)*)
U28A 2-                          BEGIN
U28A --                          ERROR(RF@,TF@);
U29E --                          PUT(RF);
U2AA --                          GET(TF);
U2B6 --                          Q:=TRUE;
U2BC --                          WHILE NOT EOF(TF) AND Q DO
U2DU --                            IF ISEQ(TF@.TNUM,RF@.RNUM) THEN GET(TF)
U2F8 --                                                       ELSE Q:=FALSE;
U3U6 -2                          END
U3U6 -1     END;
U3UA --     RESET(RF);
U316 --     WRITELN(INTESTAZIONETABELLA);
U336 --     WHILE NOT EOF(RF) DO
U346 1-       BEGIN
U346 --       IF RF@.RI=STANDARD THEN
U350 2-       BEGIN
U350 --       WRITELN;
U35C --       WRITELN('*    ',RF@.RNUM.PREC:1,' ',RF@.RNUM.CARD:5,'    *    ',
U3A6 --                 RF@.SALDOIN:11:2,'    *  ',RF@.SALDOFIN:11:2,'    *     ',RF@.NDEP:5,
U3EE --                 '    *    ',RF@.NPREL:5,'    ')
U41A -2       END
U420 --       ELSE
U424 2-       BEGIN
U424 --       WRITELN;
U43U --       WRITELN('*    ',RF@.RNUM.PREC:1,' ',RF@.RNUM.CARD:5,'    *',ERRORMSG:62);
U494 -2       END;
U494 --       GET(RF)
U4AU -1       END;
U4A4 -U     END.

*HONEYWELL LEVEL 62 PASCAL COMPILATION CONCLUDED *

*NO ERRORS DETECTED IN PASCAL PROGRAM *
```

Fig. 8: (Continuazione)

```
NUMERO CONTO *  SALDO INIZIALE  *  SALDO FINALE  *  DEPOSITI  *  PRELIEVI
   C    30   *     725000.00    *    760000.00   *       1    *       1
   B    31   *LA TRANSAZIONE CON NUMERO INDICATO NON HA CONTO CORRISPONDENTE
   B    40   *     850000.00    *    894000.00   *       1    *       0
   A    10   *     270000.00    *    277500.00   *       1    *       1
   A    12   *LA TRANSAZIONE CON NUMERO INDICATO NON HA CONTO CORRISPONDENTE
```

Fig. 9: Output di un'esecuzione

procedura o la funzione si applica divenga anch'esso un parametro. Quindi ad esempio, se n è di tipo *num*, la

vfun iseq(p:num) **returns** (b:boolean)

è resa in Pascal dalla

function iseq (n,p: num): **boolean**

- la funzione *ismaj* non compare nel programma Pascal in quanto, nella traduzione dell' **if...fi** nella successione di **if...then...else...** innestati, la condizione dell'ultimo "else" diviene implicita;
- la funzione *sort* sul tipo *transaction-file* è omessa in quanto si suppone di effettuarne l'ordinamento utilizzando il SORT di sistema da eseguire prima del programma;
- la parte finale del codice è introdotta nel programma Pascal per stampare il file di report utilizzando il file standard *output*.

È inoltre opportuno far seguire l'esecuzione del programma dall'esecuzione di una utility di sistema che effettui il renaming del file afnew, attribuendogli il nome af, in modo che una successiva esecuzione del programma ritrovi in af l'ultima versione aggiornata dell'archivio conti correnti.

La fig. 8 mostra il programma Pascal.

La figura 9 mostra l'output di una semplice esecuzione che, come si può vedere dalla figura, contempla il caso in cui ad un numero di conto corrisponde una o più di una transazione e quello in cui ad una transazione non corrisponde nessun conto corrente.

Può essere utile segnalare che i due errori che si sono riscontrati all'atto dell'esecuzione del programma dipendevano entrambi da una sottovalutazione delle difficoltà insite nel passaggio dal disegno alla programmazione.
In altre parole non è stato scoperto alcun errore nel disegno del sistema, e corrette sono pure risultate la strattura del programma principale, delle procedure e le loro interfacce, così come progettate nelle macchine M1, M-num e M-altro. È invece emersa una eccessiva leggerezza, cioè una carenza di formalismo, nel passaggio dal linguaggio di disegno al linguaggio di programmazione.
Tale lacuna deriva dal fatto che tale passaggio è scarsamente guidato da WELLMADE. Esso infatti da una parte suggerisce di inserire considerazioni relative a specifici problemi di implementazione nella documentazione della macchina a cui si riferiscono. Ad esempio, nel caso in esame, la documentazione di M-num andrebbe completata inserendo l'avvertenza di non applicare a variabili di tipo *prec* gli operatori ‹ e ›.
D'altra parte, quando la "distanza" tra le strutture di dati definite negli ultimi livelli del disegno disponibili nell'ambiente di programmazione prescelto è rilevante, WELLMADE propone la definizione di ulteriori livelli gerarchici in cui le strutture di dati vengono raffinate per mezzo della "data representation". È questo ad esempio il caso quando si deve simulare con una struttura di dati statica una struttura di dati dinamica. Ma laddove non si tratti di casi così complessi, il definire nuove macchine è in generale un eccessivo appesantimento.
Nel loro insieme pertanto queste due indicazioni non garantiscano una guida efficace per passare dal disegno alla implementazione nell'ambiente hardware e software prescelto. Appare quindi necessario definire le modalità di tale passaggio in modo più efficace onde garantire il corretto completamento del disegno di un sistema.

Bibliografia

[1] A. Pizzarello, DATA REPRESENTATION IN WELLMADE, NOTE DI SOFTWARE 13/14, gennaio - giugno 1980

[2] E.W. Dijkstra, A DISCIPLINE OF PROGRAMMING, Prentice - Hall, 1976

[3] F. De Cindio, G. De Michelis, C. Simone, CONCEZIONE DEI SISTEMI EDP: IL PRIMO PROBLEMA È L'ANALISI, Sistemi e Automazione, n. 201, febbraio 1980

[4] F. De Cindio, G. De Michelis, C. Simone, GUARDED COMMUNICATING PROCESSES, Proc. Fifth Honeywell International Software Conference, Mineapolis, 1981

[5] WELLMADE: A RATIONAL DESIGN METHODOLOGY, Honeywell Information System, 1980

[6] C.A.R. Hoare, AN AXIOMATIC BASIS FOR COMPUTER PROGRAMMING, CACM 12.10, 1969

[7] S. Alagic. M.A. Arbib, THE DESIGN OF WELL - STRUCTURED AND CORRECT PROGRAMS, Springer Verlag, 1978

[8] K. Jensen, N. Wirth, PASCAL - USER MANUAL AND REPORT, Springer Verlag, 1974

UN METODO PER LO SVILUPPO DI SISTEMI SOFTWARE: IL JACKSON SYSTEM DESIGN

Lucio D'Amico (*)

Riassunto

Dopo il grosso successo ottenuto dal metodo Jackson per il disegno di programmi, che prende gli spunti principali dalla descrizione dei dati di ingresso e di uscita, Jackson si è occupato dello sviluppo di un metodo per il disegno di sistemi. L'articolo dà una sommaria descrizione delle proposte del Jackson System Design. Si parte dall'ipotesi che il lettore abbia una certa familiarità con il metodo Jackson Structured Programming che, tra l'altro, è parte del più ampio metodo per il disegno di sistemi.

Abstract

The aim of this note is to give a brief overview of the Jackson System Design method (JSD). JSD includes, as a part, the well known Jackson Structured Programming method (JSP). It is supposed that the reader is familiar with JSP concepts.

1. INTRODUZIONE

Jackson System Design è una estensione del metodo JSP (Jackson Structured Programming) e comprende tutto il ciclo di sviluppo di un prodotto sfotware.

Il metodo JDS (Jackson System Design) è stato lanciato nel 1980 da Michael Jackson ed è ora in fase di prova e sperimentazione in Scozia, presso un gruppo di industrie ad opera delle quali il metodo potrebbe avere qualche modifica per arrivare a un consolidamento della proposta.

JSD comprende come sua parte integrante i concetti di JSP.

Il metodo propone di considerare inizialmente il problema proposto dall'utente; poi, insieme all'utente, si lavora per arrivare alla *descrizione logica del sistema;* quello che segue la descrizione dell'aspetto logico è la *realizzazione fisica del prodotto,* che risolve il problema sull'elaboratore.

(*) Istituto di Cibernetica dell'Universita di Milano. Il presente lavoro è stato effettuato nell'ambito del Progetto Finalizzato Informatica, Sottoprogetto P1, Obiettivo METOD.

Il metodo propone vari *passi* da seguire.

Nella presente nota verrà presentato, contemporaneamente alla descrizione dei vari passi, un esempio di sistema che gestisce delle transazioni bancarie per un conto corrente.

Sono, qui, fatte ipotesi semplificanti rispetto ai corrispondenti sistemi "reali". La descrizione del problema è la seguente.

L'apertura di un conto corrente avviene dopo avere steso un contratto firmato da banca e cliente.

Dopo un primo versamento, la banca rilascia il libretto degli assegni.

La banca mantiene una scheda personale che contiene tutte le operazioni che il cliente fa nel trimestre corrente.

Il cliente può fare operazioni di prelievo e versamento presentandosi alla banca, e si suppone che non possa fare altre operazioni.

Infine, è possibile che il cliente estingua il proprio rapporto. In caso di estinzione del rapporto, la banca non rimborsa gli interessi maturati per i giorni passati dall'inizio del trimestre, e quindi la cifra rimborsata al cliente in qualsiasi istante è il saldo corrente.

La banca ogni tre mesi emette un estratto del conto e uno scalare interessi per calcolare interessi e aggiornare il vecchio saldo aggiungendoci l'ammontare degli interessi.

I conti correnti devono essere sempre a debito per la banca: nel caso che un conto vada in rosso, la banca deve provvedere immediatamente a segnalare la situazione mandando un avviso al cliente.

Infine, gli impiegati della banca possono chiedere informazioni sulle operazioni fatte da qualsiasi cliente nel trimestre corrente.

2. I PASSI PER LO SVILUPPO DI UN SISTEMA

Il metodo fornisce una sequenza di sette passi mediante i quali l'utente, partendo dalla definizione di un certo problema, arriva alla fine ad avere la realizzazione di un prodotto software che rispecchia le esigenze delineate dal problema originario.

Il metodo fornisce una sequenza di passi da seguire, dove ogni passo ha in ingresso i risultati acquisiti al passo precedente e produce una uscita che servirà al passo successivo (eccezion fatta per il primo passo, che ha come unico ingresso i bisogni dell'utente, e l'ultimo passo, che fornisce come uscita il prodotto finito).

Tra i sette passi per lo sviluppo di un sistema software, i primi due servono per comprendere il mondo reale degli utenti, nei successivi tre si costruisce un modello del mondo e si definisce, oltre alla struttura del modello, un insieme di funzioni che operano su di esso. Gli ultimi due passi sono rivolti alla realizzazione fisica del prodotto sull'elaboratore.

Più in dettaglio:

il *primo passo* permette di definire le "entità/azioni": bisogna stabilire quali sono i tipi di entità che interessano il micromondo sotto osservazione e quali sono le azioni che operano sui diversi tipi di entità.

Il *secondo passo* consiste nel costruire la struttura dei tipi di entità: stabilire l'ordine in cui i tipi di entità subiscono o compiono le azioni che sono considerate significative.

Il *terzo passo* si occupa della sincronizzazione e del coordinamento tra il mondo reale e il modello.

Il *quarto passo* serve per costruire i collegamenti che possono esserci tra i vari tipi di entità.

Al *quinto passo* si aggiungono le "funzioni": si evidenzia cosa dovrebbe fare il sistema e come si possono creare gli output del modello.

Il *sesto passo* è quello in cui si costruiscono i vincoli temporali necessari per fare funzionare il sistema in modo corretto.

Il *settimo passo* si occupa della realizzazione, cioè di come si trasforma il sistema affinchè possa essere eseguito in modo efficiente sull'elaboratore.

3. IL PRIMO PASSO

Il primo passo serve per capire quali sono le mosse preliminari per arrivare alla formulazione di un modello che rispecchi le caratteristiche della realtà che si intende considerare.

Il primo passo dello sviluppo è quello di capire il mondo reale degli utenti.

Si identificano i *tipi di entità* del mondo, per esempio cliente, fornitore, ecc. e le relative *azioni*, per esempio pagamento, versamento, ecc.

Ogni nome (parola sostantiva) che l'analista incontra nella descrizione del mondo dell'utente individua un possibile tipo di entità. (Molti dei nomi che sono candidati tipi di entità verranno poi scartati perchè non risulteranno significativi per il modello che si vuole costruire).

In seguito, si vedrà che, per ogni tipo di entità individuato, esisterà un file, e l'insieme di questi files rappresenta la struttura portante del sistema.

Le azioni sono invece i verbi che si incontrano nella descrizione del mondo dell'utente: un cliente cambia indirizzo, un ordine viene modificato, un abbonamento viene chiuso.

A volte di possono incontrare più nomi di tipi di entità lessicalmente diversi, ma che non sono altro che sinonimi, in questi casi se ne sceglie uno.

Esempio

Lo sviluppo del primo passo per il problema proposto è di questo tipo: leggendo attentamente il testo, si possono identificare dei tipi di entità e delle azioni che sono significative per la costruzione di un modello.

Come tipi di entità si evidenziano subito il cliente, la banca e l'impiegato. Come azioni si identificano l'apertura del conto, il versamento iniziale, versamenti, prelievi, la chiusura del conto, l'emissione degli estratti conto, l'emissione degli scalare interessi, l'emissione di avvisi, le richieste degli impiegati.

Si può costruire una tabella come quella indicata in fig. 1, nella quale si evidenzia, per ogni tipo di entità, quali sono le azioni che la interessano e, per ogni azione, a quali tipi di entità partecipa.

ENTITÀ	AZIONI
CLIENTE	FARE L'APERTURA DEL CONTO FARE IL VERSAMENTO INIZIALE FARE VERSAMENTO FARE PRELIEVO FARE CHIUSURA
BANCA	EMETTERE ESTRATTO CONTO EMETTERE SCALARE INTERESSI EMETTERE AVVISI
IMPIEGATO	FARE DELLE RICHIESTE

Fig. 1

In genere, una azione è attiva (per esempio, il cliente fa il versamento) ma, a volte, come si vedrà procedendo nello sviluppo dell'esempio, possono esserci delle azioni che le varie entità subiscono.

	Cliente	Banca	Impiegato
APRIRE	F		
VERSARE (1°)	F		
VERSARE	F		
PRELEVARE	F		
CHIUDERE	F		
EMETT. EST. CONTO		F	
EMETT. SCAL. INT.		F	
EMETT. AVVISI		F	
FARE RICHIESTE			F

Fig. 2

La F nella tabella di fig. 2 significa che l'entità della colonna fa l'azione che si trova nella corrispondente riga. Se ci fosse una S significherebbe che l'entità di un certo tipo subisce l'azione indicata.

4. IL SECONDO PASSO

Alla fine del primo passo si ha un elenco di tipi entità e azioni. Inoltre può essere comodo avere costruito una tabella che indichi quali azioni agiscono su ciascun tipo di entità e quali tipi di entità collaborano a realizzare una azione. Scopo del secondo passo è quello di definire la struttura di ciascun tipo di entità definendo, in pratica, il suo ciclo di vita.

Nel secondo passo si definiscono le *strutture dei tipi di entità* identificati.

Si definiscono, se possibile, anche delle *relazioni temporali* fra le azioni e la vita dell'entità.

Le strutture sono costruite usando le tre componenti base di sequenza, selezione e iterazione già note da JSP.

Anche in questo passo ci sono delle interrelazioni con l'utente, e può essere utile che quest'ultimo comprenda gli schemi della strutture dei vari tipi di entità proposti dall'analista.

Lo scopo fondamentale della struttura è quello di spiegare quali sono le relazioni tra i tipi di entità e le azioni.

È importante, in una fase iniziale del progetto, costruire degli schemi di struttura che siano semplici e senza ridondanze, ma contengano tutte quelle caratteristiche che si possono considerare importanti per un dato tipo di entità.

Esempio

Nell'esempio che si sta trattando le strutture sono rappresentate in fig. 3.

Conviene qui esaminare la struttura di "cliente", che è la più completa. Il significato è che il cliente, nella sua vita rispetto a un conto corrente, deve, come prima operazione, compiere una apertura, poi fare un versamento iniziale che può essere seguito da una iterazione di operazioni (dove ogni operazione deve essere o un versamento o un prelievo) e, infine, deve effettuare l'operazione di chiusura del conto.

Si notano le tre strutture basiche di sequenza, selezione e iterazione.

A questo punto, dopo avere esemplificato il concetto di struttura, si può affermare che il tipo di entità "banca" si può trascurare: infatti si nota che essa non fa che produrre delle uscite e Jackson propone che tipi di entità senza rilevanti ingressi vengano inizialmente trascurati (le loro uscite verranno espresse come funzioni nel quinto passo).

Naturalmente, sarebbe già stato possibile eliminare il tipo di entità banca alla fine del primo passo: nell'esempio è stata eliminata solo alla fine del secondo passo per esemplificare meglio il concetto di struttura. (In realtà il tipo di entità banca dovrebbe avere un ingresso che segnala la scadenza del trimestre, ma si tratta di un ingresso poco frequente, per il quale sarà sufficiente introdurre un temporizzatore nel sesto passo).

5. IL TERZO PASSO

Lo scopo del terzo passo è quello di collegare le strutture create al secondo passo. Si utilizzano due concetti fondamentali per il metodo: quello di coordinamento e quello di sincronizzazione.

Al termine del secondo passo, il modello consiste di un insieme di strutture che sono collegate tra di loro e che, pure simulando il comportamento della realtà quando trattano un evento, non hanno degli ingressi che permettano loro di "accorgersi" del fatto che tale evento sia avvenuto nella realtà.

A tal fine sono necessarie due ulteriori attività da parte dell'analista: la *sincronizzazione* e il *coordinamento.*

Al fine di sincronizzare il modello con il mondo reale, nel momento in cui si fa una operazione nel mondo reale si vuole che il modello se "ne accorga": si deve, cioè, operare una sincronizzazione tra gli eventi del mondo reale e le azioni del modello.

Per questo potrebbe essere sufficiente inviare un segnale privo di informazioni al modello, ogni volta che si verifica un evento qualsiasi nel mondo reale. Attraverso questo segnale si può dire che il modello viene "risvegliato" (fig. 4).

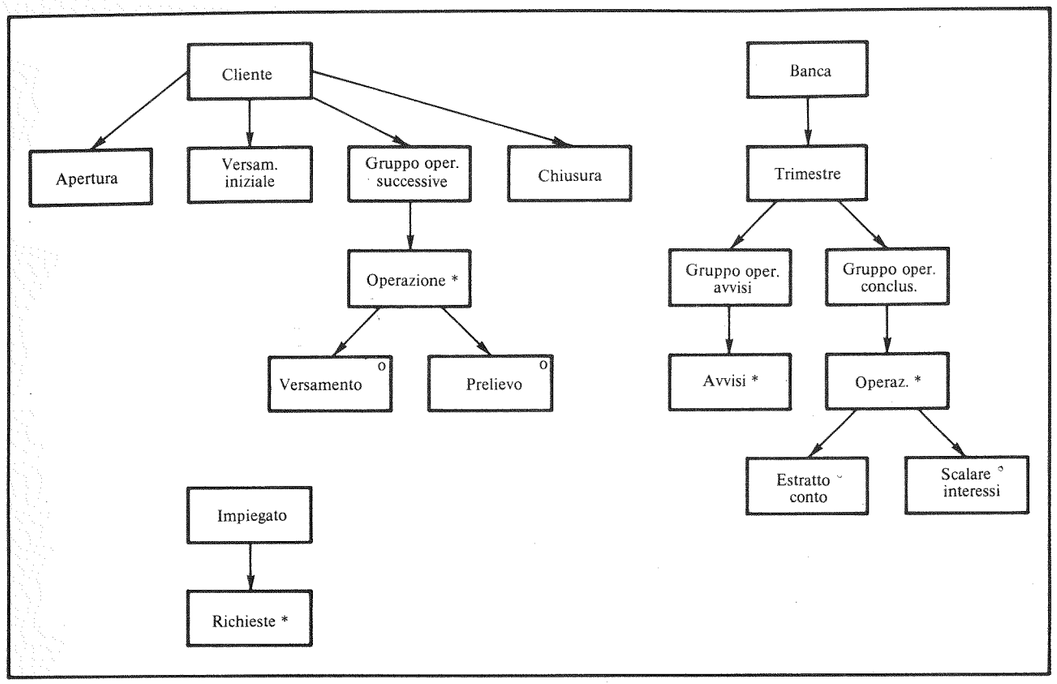


Fig. 3

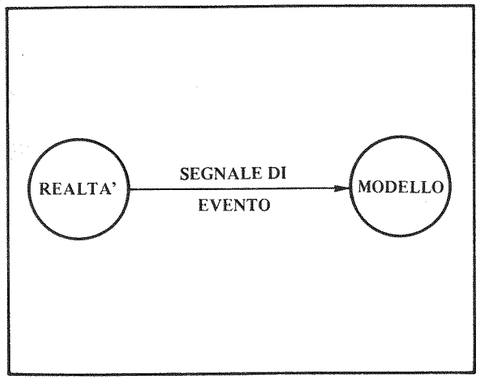


Fig. 4

Questo però non è sufficiente: al modello occorre, infatti, segnalare anche "quale" evento è occorso nella realtà.

Coordinare significa, pertanto, che quando avviene un evento nella realtà, al segnale di sincronizzazione deve essere aggiunta una informazione che permetta al modello di intraprendere quelle azioni che portano ad una simulazione corretta della realtà.

In tal modo se un modello consiste di un insieme di n strutture, ciascuna delle quali identifica, seguendo i vari percorsi, diverse azioni da intraprendere a seconda dei differenti eventi che possono succedere, l'identificazione dell'evento occorso nella realtà permetterà di scegliere il cammino da percorrere per effettuare la simulazione dell'evento stesso.

Dal punto di vista elaborativo, il coordinamento e la sincronizzazione possono essere realizzati per mezzo di un file che invia, a quel processo che implementa la struttura coinvolta da un certo evento, un record contenente informazioni relative all'evento occorso.

Naturalmente ci sarà sempre un certo ritardo tra l'azione avvenuta nel mondo reale e il suo trattamento da parte del processo di elaborazione.

Esempio

Nell'esempio, il terzo passo (sincronizzazione e coordinamento) dà come risultato lo schema di fig. 5.

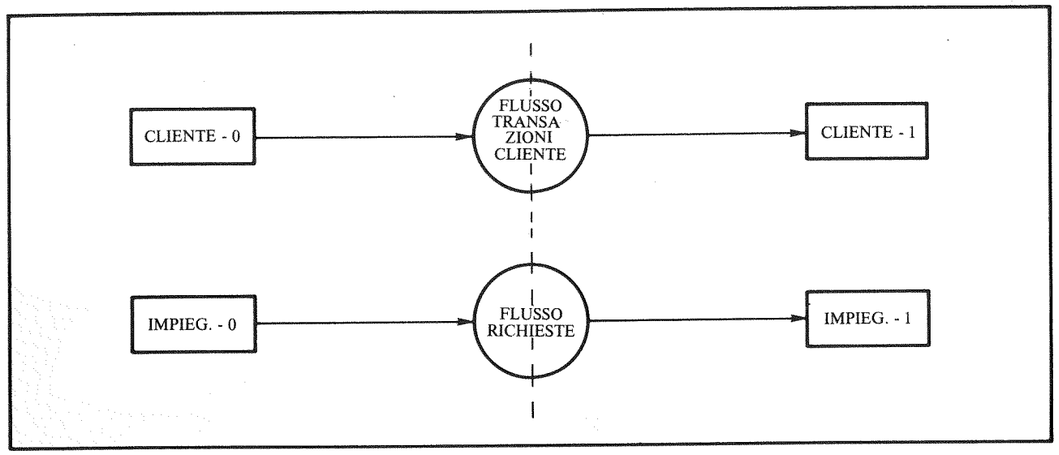


Fig. 5

Sono stati introdotti due file che sincronizzano e coordinano cliente e impiegato del mondo reale (cliente-0 e impiegato-0) con cliente e impiegato del modello (cliente-1 e impiegato-1).

È opportuno notare che cliente-1 è la struttura che tratta gli eventi di apertura, versamento, prelievo e chiusura di conto. Quindi il flusso transazioni cliente segnalerà alla struttura, nel momento in cui occorre un evento, se si tratta di una apertura, di un versamento, di un prelievo o di una chiusura di conto. In base a questa informazione, il processo attraverso il quale, come vedremo più oltre, si realizza la struttura cliente-1, sarà in grado di eseguire azioni che simulino le corrispondenti azioni avvenute nella realtà.

La linea tratteggiata, sempre in fig. 5, indica la separazione tra il mondo reale e il modello, e si vede che i due flussi fanno proprio da trait-d'union tra i due mondi.

Per concludere il terzo passo, l'analista darà una descrizione della struttura dei flussi introdotti, con i metodi proposti da JSP. Questa descrizione non viene data nel presente articolo, in quanto si presume la conoscenza del JSP da parte del lettore.

6. IL QUARTO PASSO

Per introdurre il quarto passo, che si occupa di collegare le varie entità del modello tra di loro in modo da evidenziare eventuali relazioni, è necessario iniziare la discussione dal concetto di vettore di stato.

È necessario ora anticipare, senza entrare nei dettagli, qualche concetto che sarà espanso nel seguito, quando si parlerà di realizzazione.

Nel momento della realizzazione, succederà che i tipi di entità e le strutture che modellano le loro azioni si trasformeranno in un insieme di processi e di flussi di dati: nel caso più semplice, ad ogni tipo di entità corrisponderanno un processo e un flusso di informazioni.

In una fase iniziale non si opera ancora questa scomposizione dei tipi di entità in processi e flussi di informazioni, ma ogni tipo di entità è visto come un tutt'unico. Sempre in fase logica, ad ogni entità di un certo tipo si associa il cosiddetto *"vettore di stato"*, che contiene la storia delle informazioni "importanti" relative alla entità.

Le scelte di quali informazioni rappresentative debba contenere il vettore di stato di una certa entità verranno effettuate dall'analisi in base a criteri difficili da generalizzare, anche se si possono indicare delle linee guida.

Anzitutto, nella fase precedente alla realizzazione, dal punto di vista concettuale ad ogni entità di un certo tipo Jackson propone che si associ un processo separato che la tratta; ad esempio, se ci fossero centomila entità cliente ciascuna avrebbe in fase logica un suo processo che la tratta.

Può, per esempio, avvenire che una certa entità inizi a essere simulata dal corrispondente processo, ma che, a un certo punto della simulazione, il processo venga sospeso per attendere che si verifichi un evento. In seguito al verificarsi dell'evento, il processo dovrà proseguire l'esecuzione non più dall'inizio, ma dal punto al quale era arrivato prima della sospensione. Per realizzare ciò, è necessario conservare memoria dello stato del processo associato a ciascuna delle entità.

Il vettore di stato si associa pertanto a ogni entità reale, con lo scopo di tenere aggiornato lo stato del processo relativamente a quella entità. Oltre a queste informazioni, è molto conveniente che il vettore di stato contenga delle informazioni attorno alla storia della corrispondente entità, cioè una certa sequenza di eventi occorsi a quell'entità.

A volte può essere impossibile memorizzare nel vettore di stato tutta la storia di una entità. Si può per esempio pensare al caso del tipo di entità cliente: esso potrebbe effettuare nella sua vita un numero di transazioni grandissimo e, poichè il vettore di stato deve essere di veloce consultazione (è la struttura più consultata perchè è quella che contiene le informazioni fondamentali), in esso si rappresenteranno soltanto dei valori riepilogativi.

Questi valori saranno, ad esempio, dei totali accumulati, o dei valori chiave di costante uso nell'elaborazione.

A questo punto si può passare alla descrizione vera e propria del quarto passo.

Il quarto passo ha lo scopo di collegare le strutture dei tipi di entità tra di loro. Questo collegamento è spesso costruito attraverso i vettori di stato, ossia ci si può accorgere che un certo tipo di entità necessiti, per operare, di informazioni che sono parte di un altro tipo di entità. In questo caso il processo che realizza il primo tipo di entità andrà a consultare i vettori di stato dell'altro tipo di entità.

Esempio

Nell'esempio, è sicuro che "impiegato-1" dovrà esaminare i vettori di stato delle entità cliente nel momento in cui richiede informazioni sullo stato attuale di un certo cliente (ad esempio se ne richiede il saldo attuale) (fig. 6).

Il collegamento ◇ significa che viene esaminato un vettore di stato !

La direzione ↓ significa che il flusso di informazioni va da cliente a impiegato, è dunque l'entità impiegato ad esaminare il vettore di stato di una entità cliente.

È utile fare alcune osservazioni sui vettori di stato dell'esempio. Di norma, se nella realtà ci sono n clienti e m impiegati si dovrebbero avere n+m vettori di stato.

Esaminando però l'entità impiegato, si nota che questa non possiede una storia che è necessario memorizzare: le richieste di informazioni da parte dell'impiegato (ad esempio sul saldo di un certo cliente) vengono subito soddisfatte dal sistema e non è necessario tenerne traccia.

Inoltre, non ha importanza quale sia l'impiegato che effettua una richiesta, perchè il comportamento del processo che produce la risposta è identico chiunque sia l'impiegato.

Dunque non è necessario creare un flusso di vettori di stato associati agli impiegati.

Se nel modello si è posta l'entità "impiegato", è solo perchè tale entità necessitava di una sincronizzazione e di un coordinamento con la realtà. Ma l'entità impiegato, nella realizzazione, si trasformerà solo in un processo.

Si è invece già visto che è necessario un vettore di stato per ogni cliente.

Quanto al contenuto del vettore di stato di ogni cliente, è utile aspettare a dargli una struttura fino alla fine del quinto passo, nel quale si definiranno le funzioni.

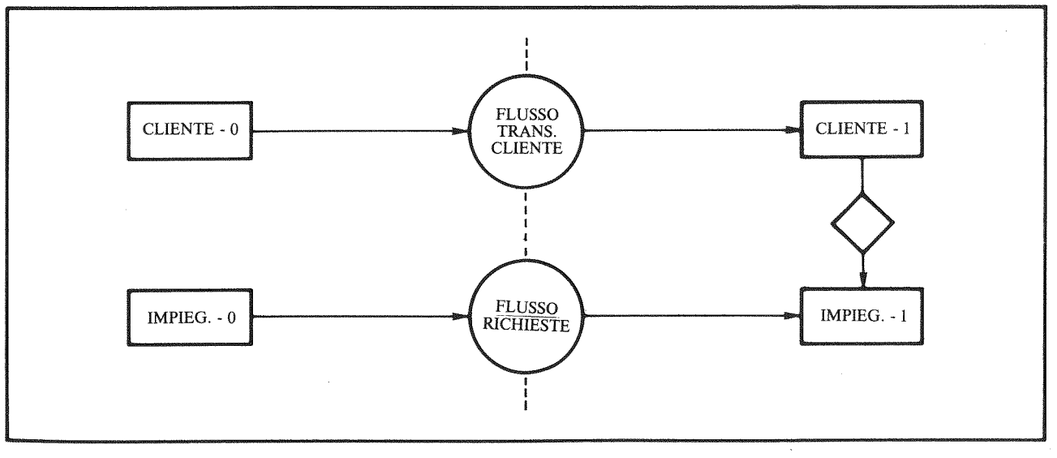


Fig. 6

7. IL QUINTO PASSO

Fino al quarto passo non ci si è occupati di quali sono le uscite del sistema. Nel quinto passo introducendo il concetto di funzione si generano le uscite del sistema.

Il quinto passo ha lo scopo di aggiungere le *funzioni.*

Il sistema, a questo punto, non ha uscite; gli output vengono prodotti solo attraverso l'esecuzione delle funzioni.

Jackson divide i tipi di funzione in tre classi di complessità, e propone di aggiungerle al modello partendo dalle più complesse per poi arrivare alle più semplici.

Nel caso più complicato, l'aggiunta di una funzione richiede la creazione di uno o più processi nuovi, i quali producono uscite che sono acquisite in ingresso dai processi che realizzeranno la struttura dei tipi di entità. Tali tipi di funzioni sono denominate *interattive.*

Nel caso di complessità intermedio, le richieste della funzione che si sta per creare sono di dati che si possono inserire nei vettori di stato delle entità di un certo tipo. In questo caso si creano uno o più nuovi processi le cui elaborazioni sfruttano i dati ispezionati nei vettori di stato di uno o più tipi di entità. Tali tipi di funzioni sono dette *imposte.*

Nel caso più semplice, l'aggiunta di una funzione richiede l'inserimento delle operazioni di uscita nel testo dei processi già esistenti, che contengono già di per sè tutti i dati necessari per potere provvedere alla generazione delle uscite. A volte si deve aggiungere un semplice processo che ha il compito di raccogliere, formattare e stampare le uscite. Questi ultimi tipi di funzioni sono dette *inserite.*

Esempio

Per trattare il quinto passo nell'esempio, è necessario domandarsi quali sono le uscite che il modello deve generare.

Esse sono: 1) l'estratto conto; 2) lo scalare interessi; 3) una lettera nel caso in cui qualche conto vada in rosso; 4) le operazioni dell'ultimo trimestre (che devono essere controllabili in qualsiasi istante).

Si nota che nel modello interessano solo le operazioni che ogni cliente ha fatto nel trimestre corrente. Si può pertanto supporre che il vettore di stato contenga tutte le operazioni che il cliente ha fatto in quel trimestre e, per non perdere, ad ogni trimestre, la storia completa, il saldo iniziale e il saldo corrente. Le ultime due informazioni servono anche all'entità impiegato nel momento in cui vuole fare una richiesta sullo stato di un certo cliente e perciò si spiega il collegamento introdotto al quarto passo.

Per potere fare l'estratto conto è sufficiente che un processo estratto conto vada a esaminare il vettore di stato del cliente, perchè questo contiene tutte le informazioni necessarie; si tratta quindi di una funzione imposta.

La seconda funzione è la più complessa: infatti un processo scalare interessi deve non solo esaminare il vettore di stato di tutte le entità cliente, ma anche generare un flusso di interessi, che servono per aggiornare il saldo corrente che, in seguito, diventerà il saldo iniziale per il nuovo trimestre. La funzione creata è quindi di tipo interattivo.

La terza funzione è la più semplice perchè, contenendo il vettore di stato del corrispondente cliente il saldo corrente, la funzione è del tipo inserita.

La quarta funzione è una funzione imposta: essa deve consultare il vettore di stato del cliente per ogni richiesta fatta dall'impiegato.

Come proposto da Jackson, le funzioni si aggiungono una alla volta al modello a partire dalla più complessa.

Alla fine si ottiene una struttura del tipo di fig. 7.

Le frecce barrate significano che i processi scalare interessi e estratto conto, ogni volta che sono eseguiti, consultano il vettore di stato di tutti i clienti e non di un solo cliente.

Prima di passare all'aggiunta dei vincoli temporali, e cioè al sesto passo, si deve modificare la struttura di cliente a causa delle funzioni interattive e inserite.

La nuova struttura è del tipo di fig. 8.

Si può notare che nella nuova struttura sono state inserite delle azioni che il tipo di entità cliente non fa ma subisce: per esempio l'aggiunta di interessi al saldo attuale e il possibile avviso di conto in rosso.

8. IL SESTO PASSO

Terminato il quinto passo è terminata anche la fase di costruzione del modello. A questo punto si passa alla realizzazione fisica del sistema. Scopo del passo corrente è quello di elencare quali sono i vincoli temporali che devono essere realizzati.

Il sesto passo è quello di creare dei *vincoli temporali.* Si può facilmente chiarire il concetto di vincolo temporale con un esempio.

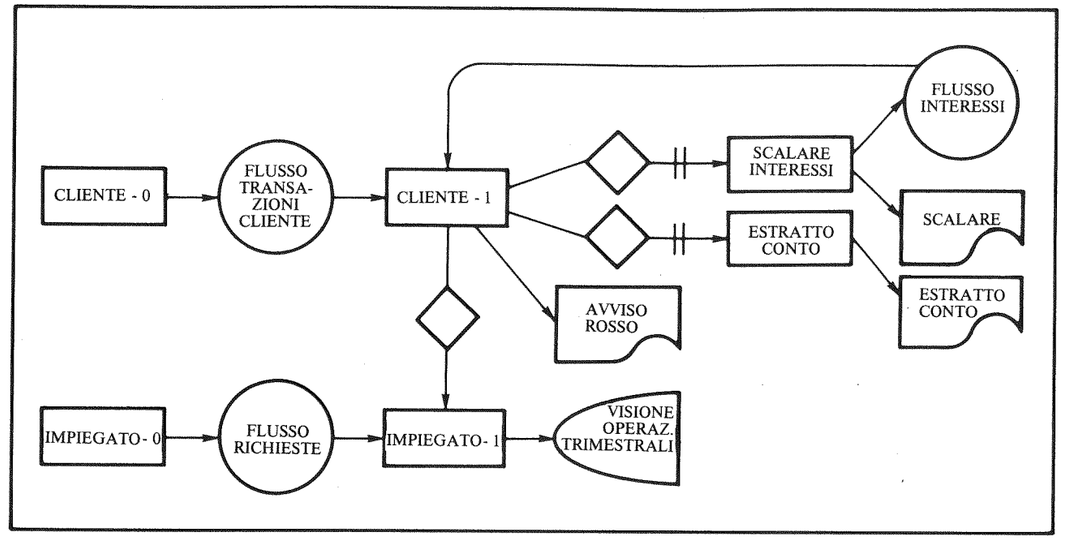


Fig. 7

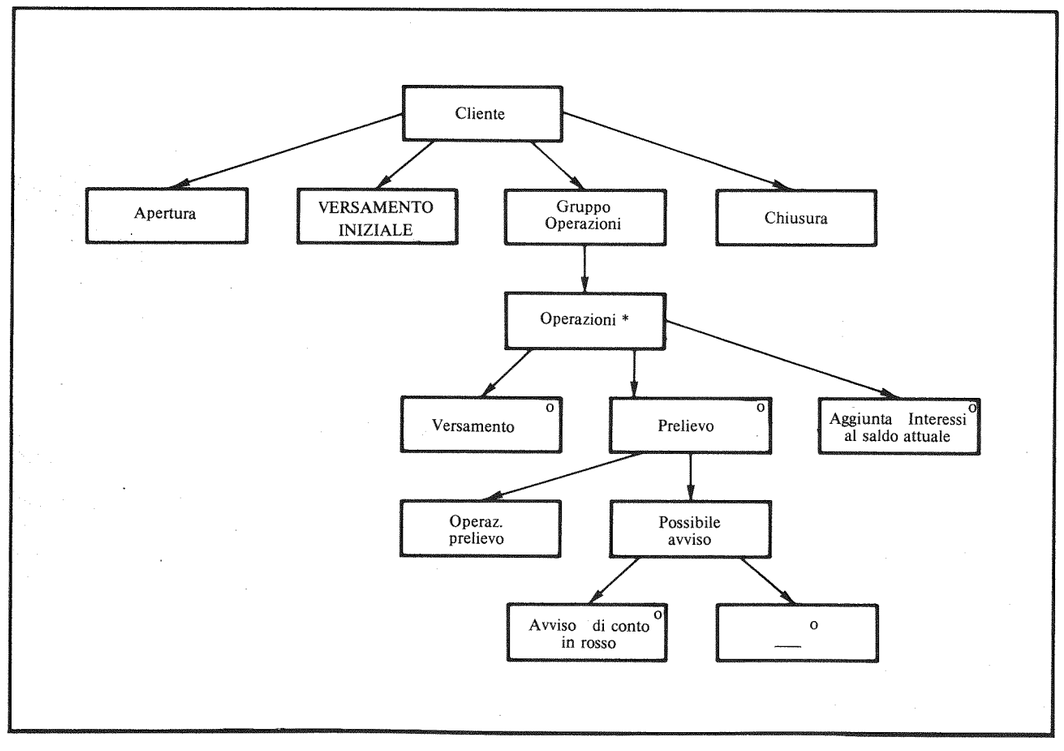


Fig. 8

Diversamente dall'esempio trattato nell'articolo, dove gli interessi sono calcolati trimestralmente, se il calcolo degli interessi dovesse essere effettuato su base giornaliera il tipo di entità cliente non sarebbe a conoscenza, nel modello, del concetto di giorno: quello che tutt'al più potrebbe conoscere è la distanza temporale tra una operazione e la successiva. D'altra parte, al termine di ogni giorno gli interessi per ogni cliente dovrebbero essere calcolati, sia che il cliente abbia fatto operazioni, sia che non ne abbia fatte. A tal fine, ci deve essere qualche cosa che avvisi che la giornata è terminata e che devono essere calcolati gli interessi.

Jackson introduce a questo proposito il concetto di Time Grain Marker (TGM). Il TGM è un indicatore della fine di un certo periodo (il TGM è realizzato con un flusso).

Nel caso sotto osservazione, alla fine di ogni giornata nel flusso TGM deve essere generato un record che evidenzia l'evento "fine giornata".

Esempio

Tornando all'esempio vero e proprio, ad ogni trimestre devono essere eseguite le funzioni di estratto conto e di scalare interessi; un altro vincolo da tenere presente è che il processo estratto conto non può lavorare fino a che il processo scalare interessi non abbia generato gli interessi, nel momento in cui aggiorna i saldi.

Dunque il modello assume la forma di fig. 9 nella quale si evidenzia l'introduzione di due TGM.

9. IL SETTIMO PASSO

È necessario introdurre alcuni concetti quali quello di Scheduler e di Inversione di programmi.

Quando si passa allo sviluppo dei programmi è indispensabile considerare il sistema fisico.

Come il compilatore trasforma il sorgente in linguaggio valido per la macchina, nello stesso modo si trasforma il sistema logico nel sistema fisico.

Sarebbe ideale avere un traduttore che trasformi automaticamente il sistema logico nel sistema fisico e in futuro non si dispera su tale possibilità. L'automazione della trasformazione si pensa che non possa essere completa, ma per lo meno si spera che si possa automatizzare dopo avere fatto la scelta del sistema fisico.

Infatti, nel momento in cui si deve passare dalle specifiche logiche alla realizzazione, ci sono varie alternative: il sistema può essere batch o interattivo, può essere basato su un file system o su un database.

Quando l'utente ha fatto questa scelta, si può pensare che, in futuro, il passaggio da specifiche logiche a realizzazione possa essere fatto in modo automatico.

Per il momento, si esaminano quali sono i passi manuali che devono essere fatti nella fase di realizzazione.

Si è già affermato che le entità di un certo tipo possono essere più di una (ad esempio, esistono più entità cliente) e che, per ogni entità, nella fase logica del progetto si pensa che esista un processo che la simula. Quindi, ogni ingresso dovrebbe idealmente entrare in un processo differente. Questo non è sicuramente realizzabile in fase di implementazione, per cui Jackson propone che tutti gli ingressi del sistema vadano in ingresso a un nuovo processo chiamato *scheduler* che si occupa del loro smistamento.

Nella fase di realizzazione oltre alla costruzione dello scheduler, (se il sistema non fornisce facilitazioni già utilizzabili), i tipi di entità (vettori di stato e struttura) vengono decomposti in un flusso di dati (che si ottiene facendo il cosiddetto scorporo del vettore di stato delle entità).

Per ogni processo, si conoscono quali sono gli ingressi e quali sono le uscite (descrizioni esatte dei contenuti dei record di ingresso e di uscita sono state fatte durante lo sviluppo delle fasi di sincronizzazione e coordinamento e di aggiunta delle funzioni).

Utilizzando quindi JSP, si costruiscono ora i relativi programmi.

(A volte, per semplificare lo sviluppo e/o migliorare le prestazioni, si può scegliere di suddividere un flusso in più flussi o modularizzare ulteriormente i programmi).

A questo punto, il sistema è praticamente costruito; ci possono però essere degli inconvenienti causati da un alto numero di flussi intermedi che non sono nè di entrata nè di uscita dal sistema. Per eliminare questi flussi si usa la tecnica di inversione di un programma rispetto a tali flussi (cfr. JSP).

La tecnica dell'inversione non è descritta in questo lavoro perchè esistono già sul mercato strumenti che effettuano l'inversione in modo del tutto automatico.

Inoltre, nella costruzione, non è indispensabile arrivare al livello del linguaggio di programmazione: arrivati allo Schematic Logic del JSP, esiste infatti uno strumento che automaticamente traduce lo schematic logic in Cobol.

Esempio

Per costruire in dettaglio lo sviluppo dell'esempio, bisognerebbe definire con esattezza le specifiche di ingresso e di uscita di ogni processo da realizzare.

È chiaro comunque che la maggior parte del lavoro di costruzione di specifiche di ingresso e di uscita, anche se, per brevità, ciò non è stato fatto, nello svolgimento di un progetto reale sarà già stato parzialmente fatto nelle fasi precedenti. (Si noti, comunque, che il metodo non fornisce un passo preciso in cui definire le specifiche di ingresso e di uscita del singoli processi).

Costruite le specifiche per ogni processo, si crea, quindi, con il JSP, il disegno dei programmi che realizzano i processi.

Guardando l'ultimo schema costruito si nota che gli ingressi al sistema sono sparsi tra i vari processi; è necessario quindi creare uno scheduler, il cui disegno è molto approssimativamente del tipo di fig. 10.

(In pratica si invertono tutti i processi che hanno degli ingressi rispetto ai flussi dei loro ingressi e delle loro uscite).

Tutti gli ingressi sono ora trattati da uno scheduler, che a seconda del loro tipo, attiva il processo interessato passandogli il record letto.

Anche tutte le operazioni di uscita (aggiornamenti del flusso rappresentante i vettori di stato o uscite reali) vengono effettuate, su richiesta dei vari processi, dallo scheduler.

Dunque si ottiene una realizzazione del tipo di fig. 11.

Si nota che c'è ancora un flusso intermedio che dà fastidio, il flusso interessi, e lo si può eliminare facendo una inversione di scalare interessi rispetto al flusso interessi.

Si ottiene la situazione conclusiva di fig. 12.

Si nota che si è utilizzata una nuova notazione: una casella del tipo di fig. 13 rappresenta lo scheduler.

Una rappresentazione del tipo di fig. 14, in cui n processi sono rappresentati sotto lo scheduler, significa che tali processi sono stati invertiti rispetto a un file e sono in relazione con lo scheduler.

Nel caso trattato, il processo clienti è stato invertito rispetto al flusso entrante, rispetto allo scorporo del vettore di stato e rispetto al flusso uscente, scalare interessi rispetto al flusso interessi. Inoltre sono state fatte delle inversioni rispetto ai TGM.

Dunque il problema conclusivo è quello di costruire un algoritmo per lo scheduler; per le inversioni il più viene in generale fatto da qualche preprocessore in modo automatico.

Infine una rappresentazione del tipo di fig. 15, che nell'esempio è presente nella struttura di fig. 16, significa che Q è stato invertito rispetto allo scorporo dei suoi vettori di stato e il flusso che rappresenta i vettori di stato è ora gestito per letture e scritture da P, anche se i processi sono sempre effettuati da Q.

Dall'esempio pare chiaro che la parte più intricata dello sviluppo e per la quale i tempi di produzione sono elevati è quella di realizzazione, per la quale è necessario introdurre potenti preprocessori per gestire in modo totalmente automatico le inversioni che, seguendo JSD, ricorrono molto spesso.

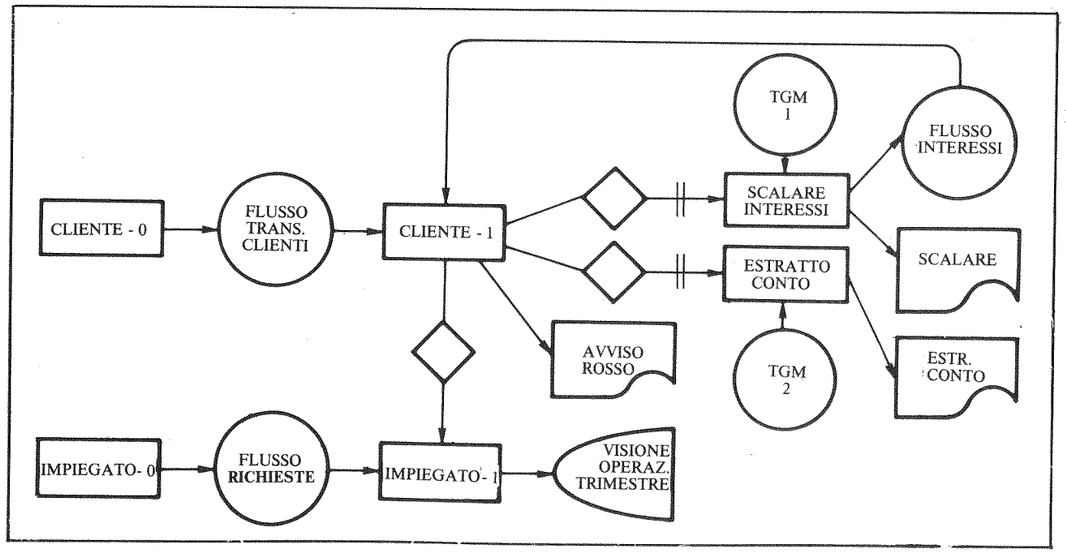


Fig. 9

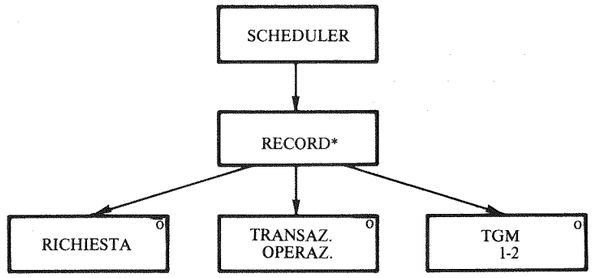


Fig. 10

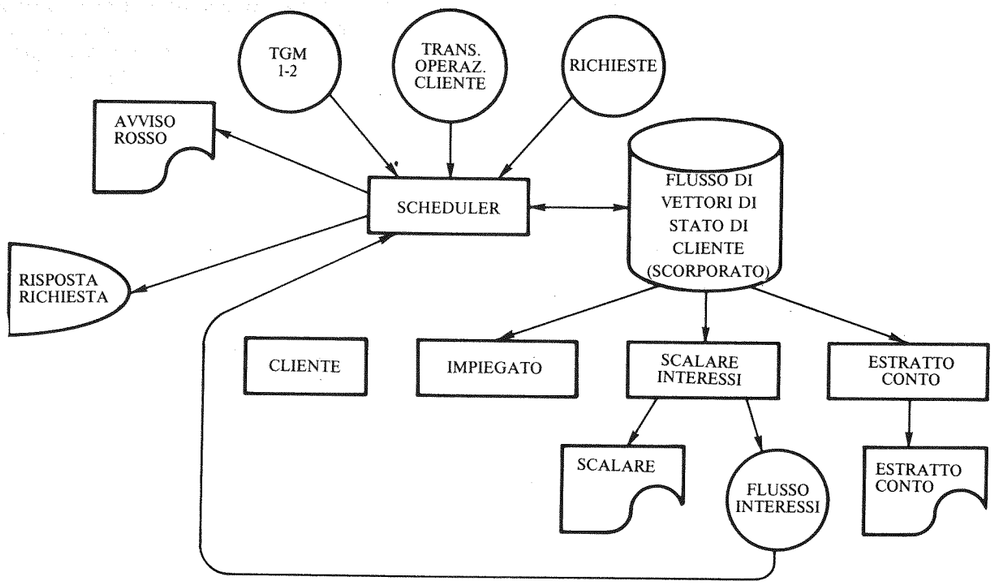


Fig. 11

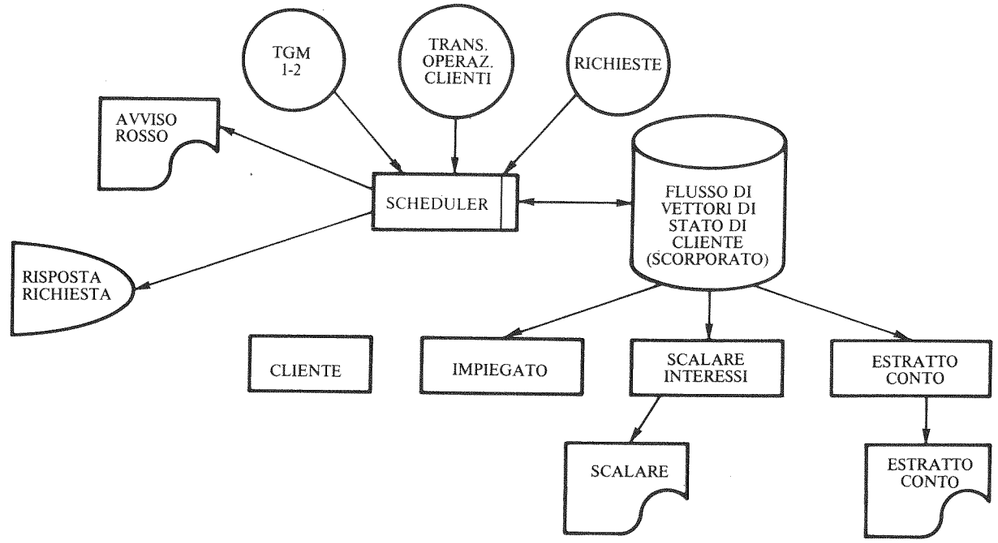


Fig. 12

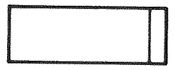


Fig. 13

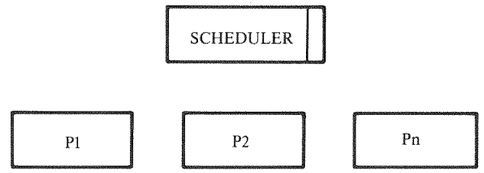


Fig. 14

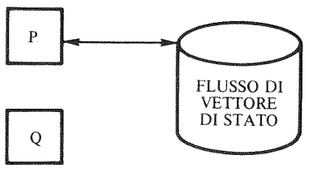


Fig. 15

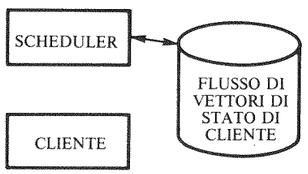


Fig. 16

10. OSSERVAZIONI

Vengono presentati alcuni concetti relativi alla standardizzazione del lavoro proposta dal metodo; si evidenziano l'importanza che il metodo dà alla separazione tra struttura e funzioni e alcuni concetti relativi alla fase di realizzazione.

Il metodo propone ai suoi utenti un *modo standard di lavoro*. Infatti, dopo avere assimilato le caratteristiche del metodo, succede che analisti diversi compongano sistemi simili usando delle strutture simili, per cui la manutenzione può essere agevolmente effettuata da persone che non hanno partecipato al progetto.

Il metodo divide il lavoro in *più passi*. La sequenzialità tra i passi proposti non è strettissima; infatti capita spesso di vedere che da un passo successivo si ritorna, per apportare dei miglioramenti, ad un passo precedente; per esempio, può capitare che nella aggiunta delle funzioni si noti l'esigenza di modificare la struttura di qualche tipo di entità, a causa di una discrepanza tra i due passi.

Il concetto ritenuto fondamentale nel JSD, che già si evidenzia in JSP, è quello di considerare gli *aspetti strutturali e funzionali* come *aspetti separati* del sistema. Nel mondo reale si identificano un numero enorme di aspetti che i sistemi informativi cercano di modellare per renderli consistenti con una successiva possibile elaborazione. Se si riesce a costruire una struttura che modella il mondo reale, allora la struttura può supportare le funzioni desiderate. Modifiche alla struttura ci sono quando cambia il mondo reale. Se non si operasse una separazione struttura-funzioni, nel caso in cui si dovessero modificare aspetti funzionali del sistema si sarebbe costretti a apportare modifiche anche nella struttura. È dimostrato da un grande numero di casi che la manutenzione dei sistemi è effettuata sugli aspetti funzionali, mentre gli aspetti strutturali subiscono modifiche solo raramente.

Un altro concetto da tenere in considerazione è che si fa una unica progettazione logica, mentre la realizzazione dipende dalle modalità con le quali si vuole che il sistema operi sull'elaboratore.

La documentazione di utente e dello sviluppo si ha contemporaneamente alla realizzazione dei vari passi. Il metodo non fornisce comunque degli standard per la documentazione di uso del prodotto.

La documentazione dell'uso del sistema deve essere conclusa nella parte tecnica, ma le principali caratteristiche devono essere già delineate dalla fase di fattibilità. Infatti solo negli ultimi due passi si decide se il sistema è interattivo o batch e se usa file system o data base.

Volendo dare una collocazione al tipo di persone che possono essere inserite nello sviluppo dei vari passi di un prodotto, risulta chiaro che gli ultimi due passi devono essere realizzati da tecnici di implementazione, mentre i passi intermedi potrebbero vedere delle interrelazioni tra tecnici del linguaggio proposto dal metodo e analisti di organizzazione.

Una preoccupazione di chi produce gli schemi generati dal quinto passo è che questi possano avere grosse dimensioni e essere molto intricati; lo schema per documentare il modello può essere a tal fine creato in modo top-down, per mezzo di raffinamenti successivi. Inoltre, quando possibile, bisogna cercare di suddividere il sistema in vari sottosistemi che abbiano poche interrelazioni tra loro.

11. BIBLIOGRAFIA

[1] M.A. Jackson, PRINCIPLES OF PROGRAM DESIGN, Academic Press, 1975

[2] M.A. Jackson, SEMINAR ON SYSTEMS DEVELOPMENTS, 1980

[3] T. Halsey, JACKSON SYSTEM DESIGN, Sistemi e automazione, anno XXVII, n. 213, marzo 1981

[4] M.A. Jackson, JSP-COBOL, AN OVERVIEW, Jackson System Limited, 1980

[5] G. Rossanigo, I CONFLITTI DI STRUTTURA NELLE METODOLOGIE CHE DERIVANO I PROGRAMMI DALLA STRUTTURA DEI DATI: LA METODOLOGIA JACKSON E UNA DERIVAZIONE FORMALE DEI CONFLITTI, tesi di laurea, facoltà di fisica dell'università di Milano, 1980-81

[6] J.W. Hughes, A FORMALIZATION AND EXPLICATION OF THE MICHAEL JACKSON METHOD OF PROGRAM DESIGN, Software practice and experience, vol. 9, 191-202, 1979

[7] M.A. Jackson, MODELLING, SEQUENCING AND TRANSFORMATIONS, Proc. 3rd International Conference on Software Engineering, 1978

[8] T. Halsey, FACENDO LE COSE SBAGLIATE: UNA FAVOLA AMMONITRICE, Sistemi e automazione, anno XXVII, n. 213, marzo 1981

AN END-USER ORIENTED TIME-SHARING SUBSYSTEM FOR DATA BASE QUERY

D. Lucarella (*)

Riassunto

In questa nota viene brevemente descritto un linguaggio interattivo di interrogazione progettato per l'accesso alle basi di dati del sistema informativo ENEL. Il linguaggio è stato ideato con l'obiettivo della massima semplicità d'uso.

Abstract

In this paper, a structured italian query language is shown. First, we present the syntax of language statements in nonformal way and after we point out some characteristics of the system implementation.

1. INTRODUCTION

ENEL, the Italian Electricity Board, has developed some applications using data-bases on H 66/80: for example, the Information Systems for power plant planning and construction, for power plant maintenance and for bidding and contract management. These information systems are used through 20 remote-batch terminals and about 150 keyboard terminals distributed all over Italy.

There are automatic procedures for storing, updating and retrieving the information, but it is also frequent the need for inquiring the data-bases in an interactive way by end-users.

So, we have decided to implement a time-sharing subsystem for data-base query on Honeywell computers (series 6000 and 66) available for use with interactive terminals.

2. SYSTEM GOALS

The main aim of the system is to provide data-base retrieval and report generation capabilities in conversational way, for non-programmer end-users.
Therefore, the language was designed with the following functional characteristics:

- Simple and easy to use with standard defaults and suitable error messages
- Completely interactive, in the sense that the system asks for missing information that are necessary to go on
- Search and selection capabilities on the records of large data-bases
- Display on-line of retrieved objects or, optionally, saving for subsequent selective printing
- Short response-time against unplanned information requests
- General purpose, so that the system is not limited to specific applications but it is suitable for interacting with any IDS structured data base.

(*) Enel: Centro Ricerca di Automatica, Milano

3. USER FUNCTIONAL REQUIREMENTS

The terminal work session starts with a request for password in order to preserve from unauthorized accesses.
Then the user is prompted to type his request with the question FUNZIONE? (function ?)

The following language statements are now available for interacting with the data-base:

1. RICERCA (search) ‹object-names› (att.1,att2 ...attn) ‹key-value›

   This statement allows the search and print (only for specified attributes) of the object with the required name and key.

2. ESTRAI(S) (select) ‹obj1› (att1 ...attn) SE (if) ‹field1› OP'value' E/O ‹field2› ..., ‹obj2› ...

   This statement allows the selection, under specified conditions, of a set of objects (obj1 ... objn) logically related to that one retrieved with the previous search operation.

It is possible to use, for selection, typical relational operators and several conditions may be chained via logical AND/OR.
The specified attributes of selected objects may be printed or saved depending on the presence of character S in the keyword ESTRAI (select).

3. ORDINA (sort) ‹obj1› (field,A/D) ... ‹objn›

   This statement is optional and, if present, must follow the ESTRAI (select) statement.
   The result is that selected objects are printed or saved after being sorted on specified fields in ascending or descending order.

4. RIPETI (repeat) ‹key-value›

   This statement allows to repeat the statements of points 1,2,3 in the same sequence supplying, every time, a different key value for the starting object.

5. STAMPA (print) ‹string› (*)

   This statement allows a selective print of information retrieved and saved at point 2.

   If the character '*' is missing, all lines with the specified string are printed, otherwise lines are printed starting from that one containing the string.

6. FINE (end)

   This statement ends the terminal session and causes the exit from the subsystem.

Every time a statement has been correctly executed the system return to the question FUNZIONE? (function) and it is available to process new submitted requests.

Every statement may be supplied completely or by iterative refinement under system prompting.

Fig. 7 contains a sample terminal session illustrating the primary functions and facilities provided by the system.

4. GENERAL DESIGN CONSIDERATION

In the system design we followed a hierarchical modular approach.
Every indipendent module may be called only from his "father" and is expected to reach normal completion every time it is executed.

If a function is interrupted by user "break" key, then a special routine takes control and simulates the completion of the function.

Using this modular approach, the software is easily modifiable for new implementations.

In figure 1, the system tree-structure limited at high levels is shown.

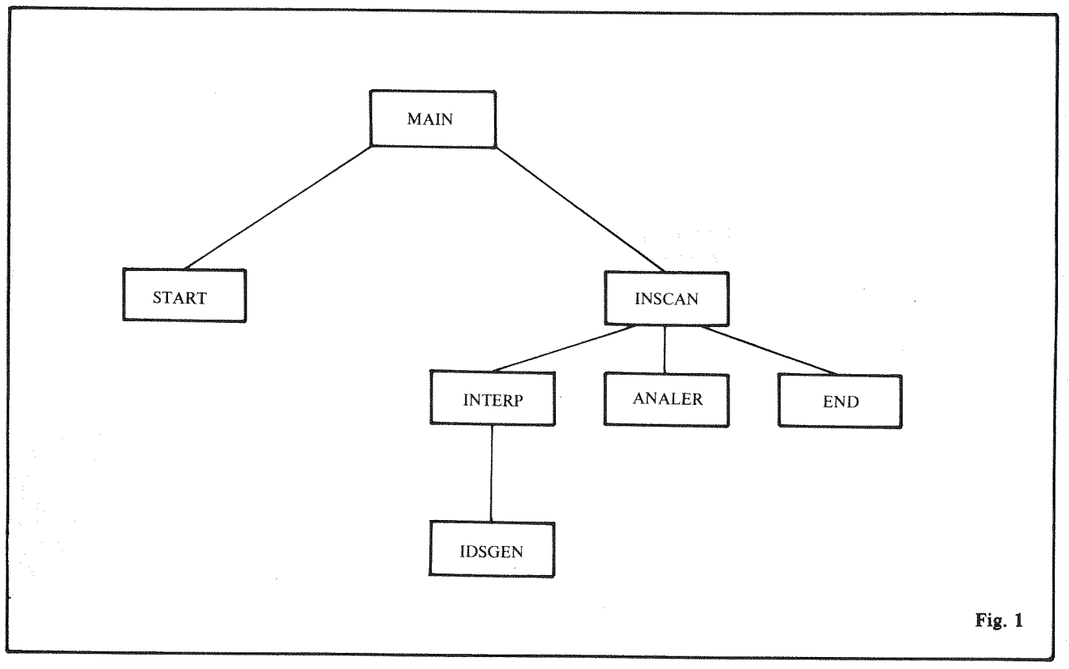


Fig. 1

A brief description of the modules follows:

- START

It is executed only once at the starting of the session. It performs password validation, files attachement and all necessary initialization operations.

- INSCAN

This module is always active. It works just like a monitor controlling I/O and dispatching other service routines.

- INTERP, IDSGEN

These two modules translate user statement and generate appropriate calling sequences to IDS library routines for data base access. Language translation processing will be detailed in the following section.

- ANALER

This module centralizes error management for the entire system. It is called when an abnormal condition occurs and analyzes a trace location set up by the interested module.

In response to the event, it may:

- ask for user correction
- force function abnormal completion sending out a diagnostic message and restarting control at function level
- force subsystem termination after printing a message.

- END

This module is called when the command "FINE" (end) is typed and provides all necessary terminating operations before closing the subsystem.

5. SYNTAX DEFINITION AND INTERPRETER

The design of language statements was intended to be as "natural" as possible from the user's point of view. Formal grammar definition was implemented using Pascal-like charts. Following this notation, syntax productions may be represented with simple diagrams.

In such diagrams, rectangular blocks represent nonterminal symbols while circular blocks represent terminal symbols, that is language key-words.
Therefore, every language statement is defined by crossing the corresponding syntax diagram.

Such a grammar is suitable for deterministic scanning and we have developed a one-pass interpreter following the so-called "recursive descending" method.
Every time a non terminal symbol is crossed than the corresponding translating procedure is called.

Semantic informations are passed as parameters of the procedures instead of expressing them in a semantic stack.
In Fig. 2, 3, 4, 5, using this formalism, we show the protocol of colloquy: from left to right, user syntax; from right to left, system syntax.

6. SUBSYSTEM FILE ENVIRONMENT

In this section, there is a brief description of the on-line files during subsystem operation.

Figure 6 shows the file environment.

- Data-base

This is the data-base permanent file. It is selected between other possible data-bases depending on the password.

- IDS-structure

It is the permanent file on which the data-base structure has been loaded. This responds to a manipulation of IDS standard software so that the IDS structure definition is standing out the application program.

Therefore, it is possible, every time, to select, load and link the appropriate structure on the basis of user's request via the password.

- Dictionary

This is a dictionary file for data-base access.

- Session file

It contains the output of those inquiries the user asks to save. There is interaction with system "current file" *SRC so that it is possible to integrate also other TSS subsystem commands.

7. FUTURE PLANS

Improvements and additions to the implemented version will be made to better fit user's requirements.
In particular, we are planning to provide other facilities in the following directions:

- to implement computing statements for simple arithmetic operations

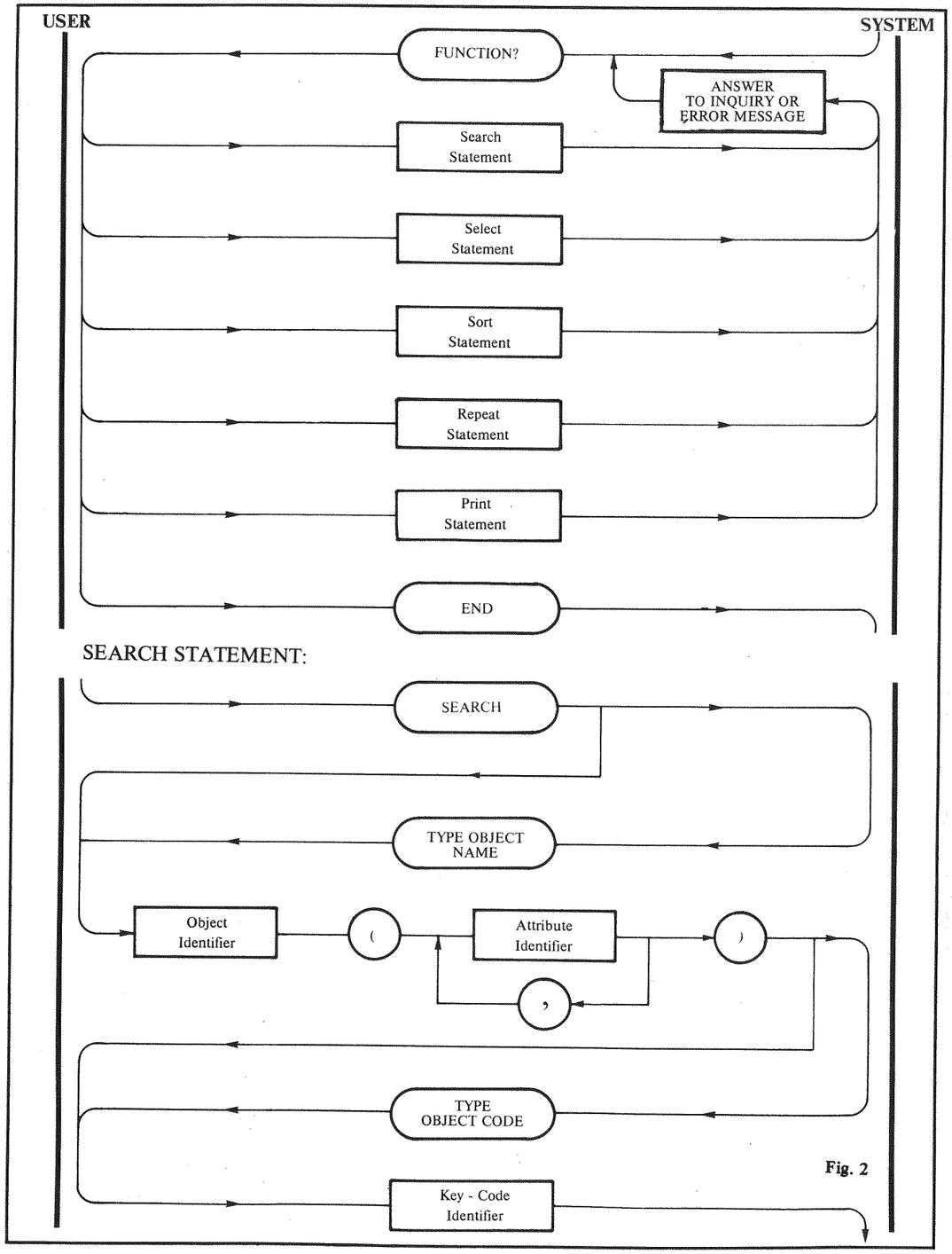

USER
SYSTEM
FUNCTION?
ANSWER TO INQUIRY OR ERROR MESSAGE
Search Statement
Select Statement
Sort Statement
Repeat Statement
Print Statement
END
SEARCH STATEMENT:
SEARCH
TYPE OBJECT NAME
Object Identifier
(
Attribute Identifier
)
,
TYPE OBJECT CODE
Key - Code Identifier

Fig. 2

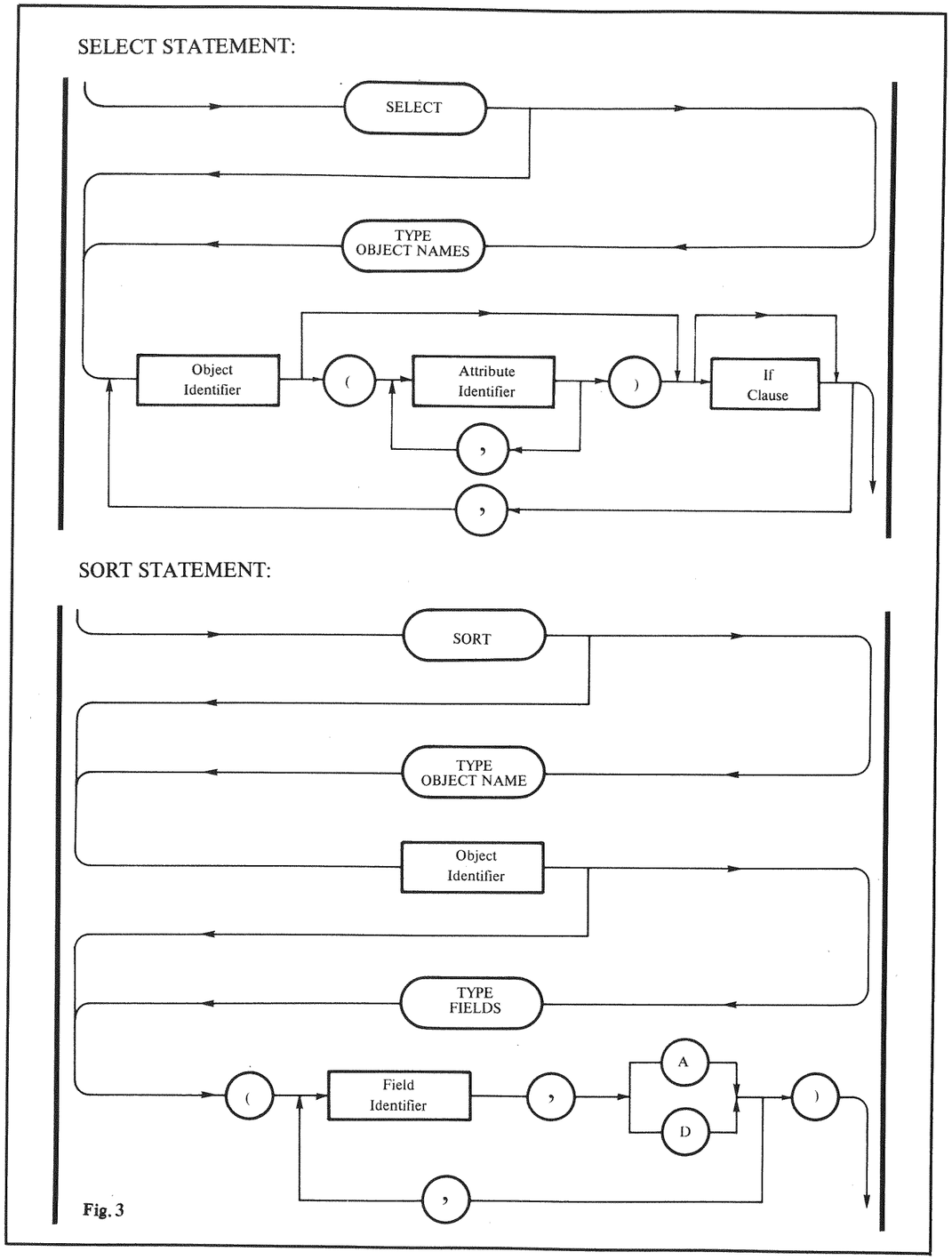
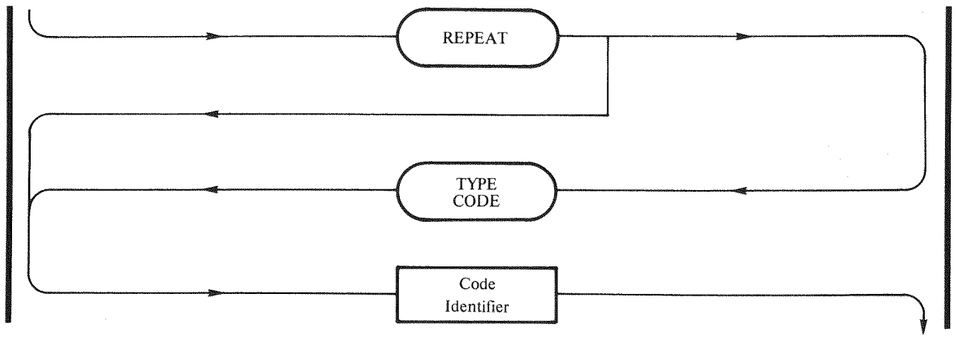
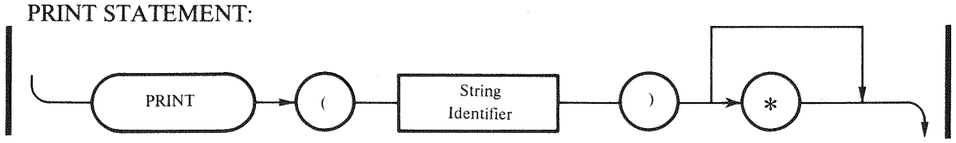

SELECT STATEMENT:
SELECT
TYPE OBJECT NAMES
Object Identifier
(
Attribute Identifier
)
If Clause
,
,
SORT STATEMENT:
SORT
TYPE OBJECT NAME
Object Identifier
TYPE FIELDS
(
Field Identifier
,
A
D
)
,

Fig. 3

REPEAT STATEMENT:



PRINT STATEMENT:



IF CLAUSE:

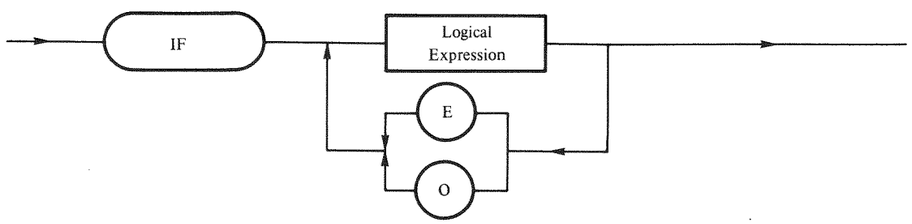
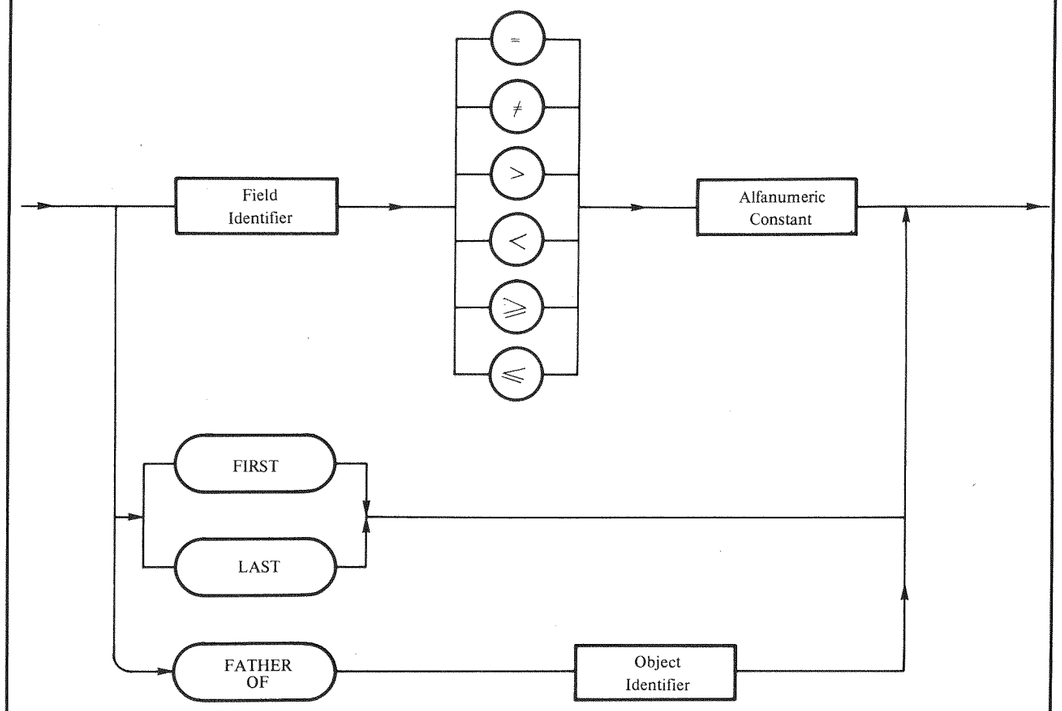


Fig. 4

LOGICAL EXPRESSION:



IDENTIFIER:

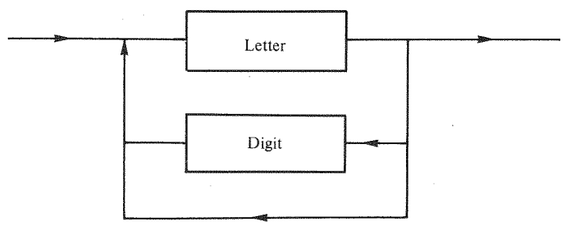


Fig. 5

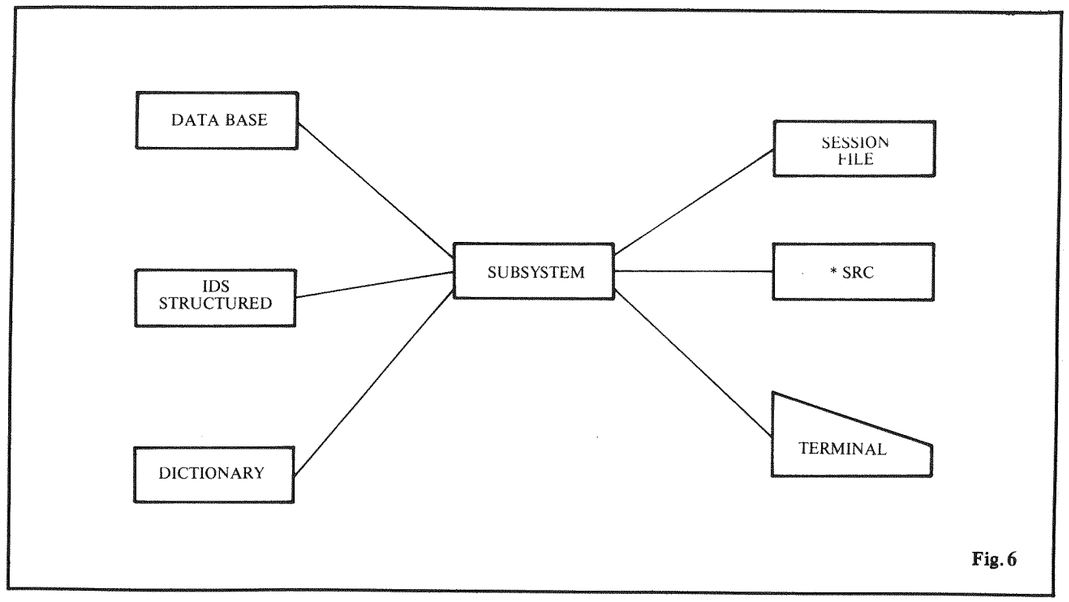


Fig. 6

- to integrate the subsystem with other TSS Text Editor Commands in order to improve output scan capability for saved inquiries
- to allows report editing with output masks set up by the user himself
- to allow on line data base update capability.

8. ACKNOWLEDGMENTS

The design and implementation of the presented subsystem has been carried out in the Computing Service of Automation Research Center, ENEL, by G. Brusati, L. Cavallari, A. Lazzati, D. Lucarella.

9. BIBLIOGRAPHY

[1] M.M. Astrahan, D.D. Chamberlin, IMPLEMENTATION OF A STRUCTURED ENGLISH QUERY LANGUAGE, Comm. ACM 18, n. 10, oct. 1975

[2] A.U. Jones, AN USER-ORIENTED DATA BASE - RETRIEVAL SYSTEM, IBM System Journal, 16, n. 1, 1977

[3] M.M. Zloof, QUERY-BY-EXAMPLE: A DATA BASE LANGUAGE, IBM System Journal, 16, n. 4, 1977

[4] D.D. Chamberlin, RELATIONAL DATA-BASE MANAGEMENT SYSTEMS, ACM Computing Surveys, 8, n. 1 march 1976

[5] D. Gries, COMPILER CONSTRUCTION FOR DIGITAL COMPUTERS, J. Wiley and Sons, Inc. New York, 1971


```
*BATTI LA PASSWORD
```

Search for object "manufacturer" (named CS) having key-code MAR.

```
*FUNZIONE?
=RICERCA CS MAR
----------------------------------------

CS05   CODICE COSTRUTTORE      MAR

CS07   NOME DEL COSTRUTTORE    MARELLI ERCOLE E C. S.P.A. - MILANO

CS08   DATA                    8NOV78
----------------------------------------
```

Select and print objects "components" if the attributes CO29 and CO35, togheter, satisfy the stated conditions.

```
*FUNZIONE?
=ESTRAI CO SE CO29='AMMAR002' E CO35;6-2<='77'
----------------------------------------

CO29   CODICE DI COMPONENTE    AMMAR002

CO30   SOTTOC.CLASSE E COST    AMMAR

CO31   SOTTOC.NUM.MODELLO      002

CO33   RIF.COSTRUTTORE         ZK-28

CO35   DATA                    12MAG77
----------------------------------------
```

Select objects "components" (named CO) printing attributes CO29 and CO35 if the field (displacement 6 length 2) of CO35 satisfies the stated condition.

```
*FUNZIONE?
=ESTRAI
BATTI I NOMI DEGLI OGGETTI
=CO(CO29,CO35) SE CO35;6-2='77
----------------------------------------

CO29   CODICE DI COMPONENTE    AEMAR002

CO35   DATA                    14GIU77
----------------------------------------
----------------------------------------

CO29   CODICE DI COMPONENTE    AMMAR001

CO35   DATA                    12MAG77
----------------------------------------
----------------------------------------

CO29   CODICE DI COMPONENTE    AMMAR002

CO35   DATA                    12MAG77
----------------------------------------
----------------------------------------

CO29   CODICE DI COMPONENTE    A6MAR001

CO35   DATA                    12MAG77
----------------------------------------
----------------------------------------

CO29   CODICE DI COMPONENTE    A6MAR003

CO35   DATA                    12MAG77
----------------------------------------
```

Repeat "search" and "select" operations supplied before, starting from the same object CS but with key-code ABA.

```
*FUNZIONE?
=RIPETI
BATTI IL CODICE DELL'OGGETTO
=ABA
----------------------------------------

CS05   CODICE COSTRUTTORE      ABA

CS07   NOME DEL COSTRUTTORE    ABA

CS08   DATA                    7MAG77
----------------------------------------
----------------------------------------

CO29   CODICE DI COMPONENTE    MBABA001

CO35   DATA                    13MAG77
----------------------------------------
```

Another search statement.

```
*FUNZIONE?
=RICERCA
BATTI IL NOME DELL'OGGETTO
=CS ABA
----------------------------------------

CS05   CODICE COSTRUTTORE      ABA

CS07   NOME DEL COSTRUTTORE    ABA

CS08   DATA                    7MAG77
----------------------------------------

*FUNZIONE?
=FINE
*
```

Fig. 7

PASCAL TOOLS

Questo spazio è dedicato a un insieme di "schede" che descrivono brevemente programmi scritti in Pascal riportati nella letteratura.

Per ogni programma viene riportato il nome, l'autore, la data, i riferimenti bibliografici, una breve descrizione desunta da tali riferimenti, e sintetici dati tecnici (ove disponibili). Viene inoltre specificato se, nei riferimenti indicati, è riportato il testo del programma.

Le schede sono classificate nelle seguenti categorie:

- UTIL: programmi di utilità di tipo generale
- TEXT: text editors e text formatters
- SYSD: sistemi interattivi di supporto alla didattica
- PGMD: programmi con finalità didattiche
- ALGO: algoritmi

(A cura di Luisa Ferrario, Istituto di Cibernetica dell'Università di Milano)

AUGMENT AND ANALYZE

UTIL

AUTORE: S. Matwin, M. Missala
Polish Academy of Sciences Comput. Centre, Pkin, Warsaw

DATA: 1975

RIFERIMENTI:
- "A simple machine independent tool for obtaining rough measures of Pascal programs", SIGPLAN Notices vol. 11, n. 11, 42-45 (1976)
- "Performance measurement of Pascal programs using Augment and Analyze", Pascal News, 12, 23-32 (1978)
- Nota su una versione Pascal DEC-10 di Augment and Analyze, Pascal News, 15, pag. 30 (1979)
- PTOOLS WRITEUP, University of Minnesota, 1977

CARATTERISTICHE:
- listing presente (641 + 224 linee)
- Pascal standard (a parte l'uso della funzione non standard CLOCK)

DESCRIZIONE:
Augment e Analyze forniscono una misura dei tempi d'esecuzione di programmi in Pascal. Augment inserisce in un sorgente Pascal, in corrispondenza delle procedure e function, il "clock-code", che in esecuzione provoca la scrittura di un file con le misure dinamiche del tempo. Analyze legge questo file e fornisce su stampante, per ogni procedure e function, il numero di chiamate e, per di ogni chiamata, il tempo d'esecuzione.

TIMELOG

UTIL

AUTORE: A.H.J. Sale

DATA: 1978

RIFERIMENTI:
- Pascal News n. 12 pag. 19

CARATTERISTICHE:
- listing presente (83 linee)
- Pascal standard, tranne poche istruzioni dipendenti dalla macchina, ben evidenziate nel listing

DESCRIZIONE:
Timelog è una procedura Pascal che scrive su un file di output (dichiarato globalmente) un log-record con la data e l'ora. Non ha parametri.

ENQUIRY

UTIL

AUTORE: A.H.J. Sale
Department of Information Sciences - University of Tasmania

DATA: 1978

RIFERIMENTI:
- Pascal News n. 13, pag. 33, dic. 1978

CARATTERISTICHE:
- listing presente (51 linee)
- Pascal standard

DESCRIZIONE:
Enquiry consente ad un programma di indagare sul proprio ambiente e di adeguarsi alle proprietà del sistema aritmetico reale. Il suo uso è particolarmente indicato per programmi che devono essere portabili attraverso diversi sistemi Pascal orientati numericamente.
Perchè la procedura operi correttamente devono essere rispettate alcune assunzioni sulla rappresentazione dei numeri reali, ben evidenziate nella documentazione raccolta.

COMPARE

UTIL

AUTORE: J. Mainer
Social Science Research Facilities Center - University of Minnesota

DATA: 1977

RIFERIMENTI:
- Pascal News, n. 12, 20-22

CARATTERISTICHE:
- listing presente (395 linee)
- Pascal standard

DESCRIZIONE:
Compare riporta in output le differenze tra due text files simili, adeguandosi alle esigenze dell'utente. Attraverso un parametro, è possibile infatti specificare il numero di linee consecutive che devono risultare uguali sui due files, per essere considerate la fine di un periodo diverso.
A causa del tempo richiesto e della memoria occupata, non è comunque consigliabile usare Compare su files con grandi differenze.

PASCUPD

UTIL

AUTORE: B.S. Carter
Department of Growth and Development - Institute of Child Health, London

DATA: 1978

RIFERIMENTI:
- "File Maintenance Using a Line Editor", Software Practice and Experience, vol. 9, 457-462, 1979

CARATTERISTICHE:
- realizzato su CDC 6600 e 7600

DESCRIZIONE:
L'autore ha sviluppato una serie di programmi che usati insieme ad un Line Editor consentono il mantenimento e l'aggiornamento di Files con record variabili.
Un programma scrive le schede di controllo per il programma di update del sistema e un altro estrae i dati del caso che si vuole aggiornare. Un terzo programma, infine, assiste i primi due riducendo la quantità di dati su cui essi operano.
La suite di programmi può essere usata con qualsiasi Line Editor nel quale le linee siano identificate da un numero di sequenza o da un particolare campo. La situazione tipica in cui risultano utili è quella per cui l'aggiornamento deve avvenire di frequente e non a tempi prefissati.

HEAPTRACE

UTIL

AUTORE: S. Schach
Weizmann Institute of Science - Rehovot, Israel

DATA: 1978

RIFERIMENTI:
- "Tracing the heap", Pascal News, n. 15, pag. 67, sett. 1979
- "A portable Trace for the Pascal Heap", Software Practice and Experience, vol. 10, 421-426, 1980

CARATTERISTICHE:
- solo documentazione
- realizzato su UNIVAC 1106 e su IBM 370/165

DESCRIZIONE:
Heaptrace aiuta l'utente nel debugging dei suoi programmi, fornendogli informazioni utili, quali il contenuto dei record nello heap. È possibile richiedere il dump di tutto lo heap oppure solo degli ultimi n record. In pratica il programma opera come un precompilatore a un passo, modificando un programma scritto in Pascal in modo da produrre il tracing dello heap, secondo le istruzioni fornite dall'utente. La base è il compilatore Pascal-P3.

PASVAL

UTIL

AUTORE: B.A. Wichmann
National Physical Laboratory, Teddington, Middlesex TW11 OLW United Kingdom

DATA: 1979

RIFERIMENTI:
- "A Pascal Validation Suite", Pascal News, n. 16, 1979 (Special Issue)

CARATTERISTICHE:
- in accordo con la nota di lavoro sul Pascal Standard BSI/ISO - Addyman (1979).
- listing presente

DESCRIZIONE:
Si tratta di una suite di programmi-test in Pascal, scritti in supporto al Pascal Standard.
Un processore Pascal viene convalidato se accetta tali programmi. Sono inoltre compresi alcuni programmi che indagano circa le caratteristiche d'implementazione e la qualità del processore.

FLOWGEN

UTIL

AUTORE: P. Roy, R. St-Denis
Department d'Informatique - Universitè de Montréal

DATA: 1976

RIFERIMENTI:
- "Linear Flowchart Generator for a Structured Language", SIGPLAN Notices, vol. 11, n. 11, 58-64, 1976

CARATTERISTICHE:
- scritto in una "...revised version" del Pascal
- realizzato su CYBER 74

DESCRIZIONE:
Da un programma scritto in Pascal, Flowgen è in grado di generare il flowchart corrispondente. A tal fine gli autori definiscono un "linguaggio flowchart" che associa ad ogni istruzione di controllo una particolare rappresentazione grafica. La loro combinazione forma un blocco e sequenze di uno o più blocchi formano un flowchart completo.
Resta il dubbio che di fatto ciò non aumenti molto la leggibilità di un programma che, scritto in Pascal, dovrebbe già essere sufficientemente strutturato e pertanto leggibile.

PASDAP

UTIL

AUTORE: M. Shimasaki et al.
Kyoto University

DATA: 1979

RIFERIMENTI:
- "An Analysis of Pascal Programs in Compiler Writing", Software Practice and Experience, vol. 10, 149-157, 1980

CARATTERISTICHE:
- realizzato su HITAC 8350

DESCRIZIONE:
Controllare il profilo dinamico di un programma Pascal è molto utile per ottimizzarne il tempo d'esecuzione. PASDAP è un preprocessore che provoca la scrittura in output del numero di chiamate, in esecuzione, di ogni procedure o function. Inoltre, con un orologio virtuale, ne fornisce il tempo d'esecuzione facendo in modo che non includa quello di un eventuale procedura chiamata.

RECOVER AND GRABBER

UTIL

AUTORE: C. Helmers

DATA: 1979

RIFERIMENTI:
- BYTE, vol. 4, n. 4, pag. 6 (1979) - editoriale

CARATTERISTICHE:
- UCSD Pascal per UCSD Pascal Systems
- listing presente (169 lin.)

DESCRIZIONE:
I programmi Recover e Grabber sono stati scritti per ricostruire il directory su disco, nel caso sia stato danneggiato. Recover è il programma che passa e controlla i blocchi su disco, cercando la parole chiave PROGRAM e stampando i primi venti caratteri che la seguono. Costruito in tal modo un directory con gli indirizzi dei blocchi fisici d'inizio di ogni programma, Grabber trasferisce i programmi dal disco danneggiato ad un nuovo disco iniziando dagli indirizzi forniti da Recover fino alla parola chiave END.

PROSE

TEXT

AUTORE: J.P. Strait
University Computer Center - University of Minnesota

DATA: 1977

RIFERIMENTI:
- "Prose Instruction Manual", Pascal News, n. 15, 35-55, 1979

CARATTERISTICHE:
- manuale completo e listing (3440 linee)
- Pascal standard (alcune riserve sulla portabilità del programma)

DESCRIZIONE:
Prose è un programma per la formattazione dei testi, nato con lo scopo di produrre documentazione "machine retrievable". Giustifica le linee e formatta il testo in pagine in modo automatico. Alcune semplici direttive servono per chiedere altre operazioni più complesse come ad esempio sottolineature, cambiamento di margini, definizione del tipo di device in output. Inoltre è possibile passare dal minuscolo al maiuscolo e viceversa, stampare solo alcune pagine del testo originale, aggiungere intestazioni e note, costruire un indice ordinato. In tutto vi sono 21 possibili direttive introducibili da tastiera in modo semplice e flessibile.
Infine il programma può ricordarsi e ripristinare l'ambiente di formattazione, cioè l'insieme di direttive precedentemente usate.

PRETTYPRINT

TEXT

AUTORE: J.F. Hueras, H.F. Ledgard
Computer and Information Science Department, University of Massachusetts, Amherst

DATA: 1976

RIFERIMENTI:
- Pascal News, n. 13, 34-45, 1978
- SIGPLAN Notices (ACM), vol. 12, n. 7, 82-84, 1974

CARATTERISTICHE:
- listing presente (1711 linee)
- in distribuzione su nastro per parecchi sistemi

DESCRIZIONE:
Prettyprint legge un file di caratteri, presumibilmente un programma in Pascal, e lo riformatta in accordo a un insieme fisso di regola di formattazione. Nato dalla convinzione che non sia necessario imporre troppe regole e opzioni nè operare una completa analisi sintattica, agisce localmente e quindi anche su frammenti di programmi. Tutti i blank e le linee blank fornite nel testo in input vengono mantenuti.

ID2ID

TEXT

AUTORE: A. Mickel
University Computer Center, University of Minnesota

DATA: 1979

RIFERIMENTI:
- Pascal News, n. 15, 31-34, 1979

CARATTERISTICHE:
- listing presente (402 linee)
- Pascal standard
- realizzato su CDC-6000, Cyber 70, 170

DESCRIZIONE:
ID2ID permette di editare velocemente e in modo accurato il testo di un programma in Pascal, sostituendo gli identificatori desiderati. Un comune text editor è generalmente adatto, poichè per ogni sostituzione sarebbe necessario passare l'intero testo del programma.
ID2ID legge il text file "SOURCE" che consiste in un programma source in Pascal, e il text file "IDPAIRS" che contiene le coppie di identificatori vecchi e nuovi. Con queste costruisce una struttura ad albero binario bilanciato (algoritmo di N. Wirth) per individuare successivamente gli identificatori durante la scansione del programma. Oltre al programma modificato viene costruito in output il file "REPORT", con i messaggi dei "warnings" e degli errori riscontrati durante l'editing. Tipicamente ID2ID potrebbe essere usato per ricodificare un programma con identificatori più lunghi e con maggior potere descrittivo.

FORMAT

TEXT

AUTORE: M.N. Condict
Lehigh University

DATA: 1975

RIFERIMENTI:
- Pascal News, n. 13, 45-58, 1978

CARATTERISTICHE:
- manuale completo e listing (1291 linee)
- esiste una versione per CDC-6000 ed una per PDP-11
- le istruzioni dipendenti dalla macchina sono ben evidenziate nel listing
- assistenza presso:
  M. Condict, Pattern Analysis and Recognition Corporation, 228 Liberty Plaza, Rome, NY 13440

DESCRIZIONE:
Format è un programma flessibile per la formattazione di programmi in Pascal sintatticamente corretti. La flessibilità consiste nella possibilità per l'utente di introdurre direttive di formattazione in esecuzione, frammiste al testo del programma. Altrimenti vengono assunte alcune semplici regole standard.
Format analizza il programma in Pascal operando un'analisi sintattica simile a quella che esegue il compilatore Pascal, con discesa ricorsiva entro dichiarazioni e statements nidificate.

VCAS

SYSD

AUTORE: Richard C. Brandt, University of Utah

DATA: 1980

RIFERIMENTI:
- Pascal News, n. 15, pag. 3, 1979

CARATTERISTICHE:
- scritto per la maggior parte in UCSD Pascal
- realizzato su TERAK 8510a
- dotato di un'interfaccia all'unità videodisco PR-7820.

DESCRIZIONE:
VCAS (Video Computer Authoring System) è un insieme di programmi per permettere ad un autore di costruire ed editare lezioni assistite dall'elaboratore, senza scrivere programmi. Le lezioni interattive includono testi, grafici, "animations" e segmenti su videotape. Per costruire una lezione così composta l'autore ha a disposizione un text editor, un graphics editor, un animation editor e un programma che seleziona segmenti di videotape. Infine con un programma assemblatore (BUILDER) crea un file che controlla la presentazione del materiale durante la lezione. La lezione vera e propria si svolge sotto il controllo di INTERP, un programma interprete che usa sia l'input dello studente che il file costruito precedentemente da Builder. È prevista un'estensione all'analizzatore delle risposte in modo che analizzi anche espressioni booleane e trigonometriche e accetti risposte di tipo diverso da quello predefinito.

XS-Ø

SYSD

AUTORE: J. Nievergelt et al.
Institute for Informatics - ETH Zürich

DATA: 1980

RIFERIMENTI:
- "XS-Ø: A self-explanatory school computer", AEDS Monitor, Apr/May/Jun 1978

CARATTERISTICHE:
- scritto in Modula, il sistema usa una versione estesa del Pascal-S sia come linguaggio d'autore che come linguaggio di programmazione a scopo didattico.
- implementato su un PDP-11/03.

DESCRIZIONE:
XS-Ø è un sistema che permette di scrivere, editare, eseguire e correggere programmi in modo interattivo, senza alcun supporto oltre a quello fornito dal sistema stesso. Comprende un graphics editor, con il quale l'utente può tracciare figure e introdurre testi, e un program editor in grado di generare automaticamente istruzioni Pascal per "disegnare" le figure. Il program editor funziona quindi come un semplice linguaggio per autore CAI, dal momento che può generare un programma Pascal che si sviluppa interamente attraverso frames prodotte dal graphics editor. Particolare attenzione è stata dedicata al dialogo tra l'utente e il sistema, nel tentativo di renderlo il più semplice possibile.

AIP

SYSD

AUTORE: K.L. Bowles - University of California

DATA: 1978

RIFERIMENTI:
- "Microcomputer Problem Solving Using Pascal", K. Bowles, Springer Verlag, 1977

CARATTERISTICHE:
- UCSD Pascal (il sistema UCSD Pascal è operativo sulla maggior parte degli elaboratori basati sui microprocessori della corrente generazione)
- i terminali collegati devono essere in grado di rappresentare su video sia testi che grafici

DESCRIZIONE:
AIP (Automated Instruction Package) è un insieme di software tools per produrre materiale CAI. L'autore ha a disposizione quattro librerie di routines precompilate in UCSD Pascal. Una contiene le routines per porre le domande agli studenti e valutarne le risposte, un'altra permette di trattare lo schermo come una composizione di diversi piccoli schermi logicamente separati, una terza consente di non includere direttamente nel programma in Pascal il testo che deve comparire su video, un'altra infine gestisce i files. Usando tali routines sono stati preparati diversi quiz automatici che illustrano il libro di Bowles indicato.

BASIS

SYSD

AUTORE: R.P. Van de Riet, R. Wiggers
Vrije Universiteit, Amsterdam

DATA: 1978

RIFERIMENTI:
- "Practice and Experience with BASIS: an Interactive Programming System for Introductory Courses in Informatics", Software Practice and Experience, vol. 9, 463-476, 1979

CARATTERISTICHE:
- scritto in Pascal per il sistema operativo Unix (4000 linee)
- supporta 20 studenti in parallelo su PDP11/45 con 248 Kbytes di memoria

DESCRIZIONE:
BASIS è un sistema interattivo per lo sviluppo di un corso introduttivo in informatica. Scritto in Pascal, è basato sull'uso del Pascal per la programmazione strutturata. Un analizzatore lessicale trasforma la linea di input in una lista concatenata di "text cells". Se si tratta di identificatori la cella contiene anche un puntatore fisso a una "idcell" e quest'ultima punta a un'altra lista che contiene gli attributi dell'oggetto al quale l'identificatore si riferisce. Il sistema comprende procedure per gestire queste liste, per leggere, stampare, controllare, eseguire e naturalmente inizializzare le procedure dell'utente.

NRSA

PGMD

AUTORE: V.W. Setzer - University of Sao Paulo

DATA: 1978

RIFERIMENTI:
- "Non-recursive Top-down Syntax Analysis", Software Practice and Experience, vol. 9, 237-245, 1979

CARATTERISTICHE:
- è presente, con commenti, il testo della procedura in Pascal standard

DESCRIZIONE:
L'autore presenta a scopo didattico una semplice procedura non ricorsiva in Pascal che opera come analizzatore sintattico discendente.
Punto di partenza è l'analizzatore sintattico discendente e ricorsivo di Wirth che accetta in input una grammatica sotto forma di grafo sintattico.
Facendo uso di tabelle, la procedura permette una completa separazione delle routines sintattiche e semantiche, risultando particolarmente flessibile.

VARI

PGMD

AUTORE: N. Wirth
ETH, Institut für Informatik, Zürich

DATA: 1979

RIFERIMENTI:
- N. Wirth, "Systematic Programming", Prentice-Hall Inc., 1973
- N. Wirth, "Algorithms+Data Structures=Programs", Prentice Hall Inc., 1975

CARATTERISTICHE:
- Pascal standard
- listing dei programmi presenti

DESCRIZIONE:
Si tratta di una collezione di programmi in Pascal di diversa complessività, da semplici esempi usati per illustrare le caratteristiche del linguaggio in corsi introduttivi, ad altri più complessi discussi in corsi su algoritmi o strutture di dati.
La raccolta mostra uno stile di programmazione che si basa sull'uso di un linguaggio di programmazione strutturato.
Nello stesso tempo costituisce un utile riferimento per alcuni algoritmi già largamente usati con una diversa formulazione.

RANDOM

ALGO

AUTORE: Brian A.E. Meekings
University of Lancaster

DATA: 1977

RIFERIMENTI:
- "Random Number Generator", Pascal News, n. 12, pag. 18, 1978
- "Random Number Generator (continued discussion)", Pascal News, n. 13, pag. 32, 1978
- "A comparison of the correlational behaviour of random number generators for the IBM 360", Whittlesey J.R.B., Comm ACM, vol. 11, n. 9, 1968

CARATTERISTICHE:
- Pascal standard, listing presente
- l'algoritmo è per sistemi con parola di 16 bit.

DESCRIZIONE:
Il generatore di numeri casuali proposto in Pascal da Meekings si basa sull'algoritmo di Whittlesey ("feedback shift register pseudorandom number generator"). Passando alla funzione un numero intero positivo diverso da zero (seed), viene restituito un numero reale nel range (0,1). La versione in Pascal è stata provata con procedure statistiche pure scritte in Pascal.

STRING

PGMD

AUTORE: Judy M. Bishop
University of Witwatersrand, Johannesburg

DATA: 1978

RIFERIMENTI:
- "Implementing Strings in Pascal", Software Practice and Experience, vol. 9, 779-788, 1979
- "Strings and the sequence abstraction in Pascal", Software Practice and Experience, vol. 9, n. 8, 671-683, 1979

CARATTERISTICHE:
- Pascal standard
- è presente il listing delle procedure compilate su AECC IBM 360/370 Pascal Compiler (350 linee)

DESCRIZIONE:
STRING è un package che concettualmente considera una stringa come un file e in pratica ne consente una realizzazione come lista concatenata di records nello heap.
Fu A.H.J. Sale che per primo propose la realizzazione di stringhe in Pascal usando il concetto astratto di sequenza in una serie di procedure avanzate. In STRING ne sono proposte altre per confrontare stringhe o convertire variabili ALFA e CHAR in stringhe, in generale più adatte per gestire stringhe brevi che tuttavia richiedono un accesso frequente e veloce.

PERFECT

ALGO

AUTORE: R.J. Cichelli

DATA: 1979

RIFERIMENTI:
- "A perfect hashing function", Pascal News, n. 15, 57-59, 1979

CARATTERISTICHE:
- Pascal standard
- listing presente (350 linee)

DESCRIZIONE:
Viene presentato un metodo per computare le seguenti funzioni hash perfette: valore hash = lunghezza chiave + valore associato al primo carattere della chiave + valore associato all'ultimo carattere della chiave.
Oltre che particolarmente semplici, sono indipendenti dalla rappresentazione dei caratteri, e consentono un ritrovamento al primo colpo in tabelle di identificatori dimensionate al minimo. D'altra parte liste con più di 45 elementi richiedono la segmentazione in sottoliste. L'algoritmo può servire per la ricerca di parole riservate nei compilatori e per filtrare parole molto frequenti nel processamento di testi in linguaggio naturale. Nella documentazione allegata sono illustrati diversi esempi.

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

Management

● J.R. Johnson, MANAGING FOR PRODUCTIVITY IN DATA PROCESSING, Q.E.D. Information Sciences, Inc., P.O. Box 181, Q.E.D. Plaza, Wellesley, MA 02181, USA, 1980, pagg. iii + 176

"It is interesting to note that ninety percent of productivity discussions emanate from the question "Why isn't Johnny coding?" But productivity is an all-encompassing topic, and individual productivity is only one of five productivity concerns in Data Processing. As a matter of fact, applying a generalized method of quantifying productivity may be more difficult at the programmer level than at other levels of responsability within Data Processing. Thus, this book addresses five aspects of Data Processing productivity in five parts: I. Individual Productivity; II. Project Development Productivity; III. Maintenance Productivity; IV. Data Processing Division Productivity; V. Data Processing Corporate Productivity. An introductory explanation precedes each part. The book is based on experiences in a large state-of-the-art DP environment. It results from pragmatic experience-doing something wrong, and then modifying the approach and trying again. In some cases, the ideas may not seem sophisticated, but simple solutions are not always obvious. [...] The following questions paraphrase those in the book and highlight reflections on productivity presented.
PART I - INDIVIDUAL PRODUCTIVITY: 1. Is Lines of Code (LOC) a good measure of individual performance? 2. How do other knowledge-type professionals monitor performance? 3. What is the number one mistake made when writing performance reviews? 4. Is "burnout" real? 5. Did programmer productivity improve during the 1970s? How much?
PART II - PROJECT PRODUCTIVITY: 1. Can detailed project phase management have a negative impact on productivity? 2. Does and advanced project management technique exist? 3. What techniques are available to estimate large projects? 4. How can the value of test time be quantified?
PART III - MAINTENANCE PRODUCTIVITY: 1. Can objectives be developed which monitor the performance of the maintenance manager? 2. Is developmental documentation intended for the same purpose as maintenance documentation?
PARTI IV - DIVISIONAL PRODUCTIVITY: 1. How many unusual abend conditions would you expect when installing a ten manyear project? 2. Why are poor standards written? 3. Should auditors have project responsibility? 4. In the year 2000, how will managers be trained?
PART V - CORPORATE PRODUCTIVITY: 1. Is data base utopia? 2. Is it more cost effective to run seventy five minis or two mainframes? 3. When computer hardware is virtually free, what will be the concern? 4. What are the future barriers to productivity? [...] The book is intended for DP professional or other managers with DP responsibilities, primarily in the commercial application environment. Also, the book is designed for use in a graduate level management computer science course."

● J.W. Cortada, EDP COSTS AND CHARGES - Finance, Budgets, and Cost Control in Data Processing, Prentice-Hall Series in Data Processing Management, 1980, pagg. xiv + 287

"Companies are spending billions of dollars each year on data processing while not always understanding why there is such a great cost or how to measure its benefits [...] This book provides advice for both data processing executives and their financial and general management on making business decisions that involve computer technology [...] It shows executives and management, in both general and specific examples, how to improve DP decision making that is relevant to a company's financial situation. For the data processing manager, it describes basic standards that a controller must consider in regard to data processing. For other types of management, the book reveals ways to measure costs and benefits in data processing and explains how some more expensive alternatives will yield greater, long-term benefits. [...] The first two chapters (intended mainly for the non-DP executive) discuss the issues of data processing trends in general and the difficult problems of applying those currents to

shortand long-term planning. [...] The next two chapters deal with the more specific issues of justifying applications in business terms, acquiring programs and equipment, and measuring costs, risks, trends, payback, and benefits. It is important that your contract negotiations support the financial strategies that you have developed for your data processing; the chapter on contracts (Chapter Five) shows you how. [...] Chapter Six describes service bureaus and facilities management as alternatives to using a data processing department within your own company. [...] Budgets, chargeout systems, controls, and accounting affect all companies in regard to data processing. Only recently have formal procedures and research into this subject area led to management involvement with data processing that is equal to its involvement with other departments within the organization. Yet data processing has its unique characteristics. Developing a good DP budget requires that certain considerations be kept in mind that do not always apply to other departments. Also, many data processing managers find that their budgets are not appreciated by non-DP executives. Chapter Seven shows specifically how to create a budget that will be relevant to upper management. An entire chapter has been devoted to vendors and consultants and a company's relationship to them, defining what can be expected from them and how their activities and knowledge can be channeled to benefit your company's management. [...]"

Indice: 1. Data Processing Technology and Trends; 2. Goals, Planning, and Data Processing; 3. Applications and Their Justification; 4. Hardware Costs and Their Justification; 5. The DP Contract; 6. Service Bureaus and Facilities Management; 7. Developing a Data Processing Budget and Controlling It; 8. Dealing with Vendors and Consultants; Appendici: Glossary of Accounting and Financial Terms; Glossary of Data Processing Terms; Select Bibliography.

Un libro sullo "Structured design"

● W.P. Stevens, USING STRUCTURED DESIGN - How to Make Programs Simple, Changeable, Flexible, and Reusable, John Wiley & Sons, 1981, pp.xvii + 213

"The primary purpose of this book is to enhance the reader's ability to make use of the concepts of structured design. It is aimed primarily at those who are responsible for developing programs. The programs to be developed may be application programs for use within the company, programs which are to be sold, operating systems, support programs, and programs that are being rewritten either completely or partially. Structured design can also be applied directly to the development of macros, cataloged procedures or EXECS, subroutine libraries, utility modules, and functions to be microcoded. It is useful wherever low development cost, ease of maintenance, flexibility, and ease of use are important. It can also be used to reduce the effort needed to produce code that must run quickly or make low use of real memory. This is possible because structured designed programs can be efficiently, rapidly, and accurately optimized. This book is intended for both experienced and beginning programmers and analysts. Throughout the book, alternatives are examined in the light of structured design concepts. [...] A basic understanding of programming concepts is assumed. Examples of code, where they are necessary, are given in relation to COBOL [...] Others who may find these concepts helpful are supervisors, managers of programmers and analysts, and programmers who will be producing or maintaining programs that have been designed using structured design [...]"

Indice: 1. About This Book; 2. Introduction to Structured Design; 3. Structured Design Concepts; 4. Structure Charts; 5. Binding; 6. Coupling; 7. Techniques and Tips; 8. An Example of Improving a Structure Chart; 9. Generating an Initial Chart; 10. A Design Example; 11. Structured Design and Performance; 12. Expanded Use of Structured Design; 13. In Conclusion.

Progettazione dei sistemi informativi

● M. Lundeberg, G. Goldkuhl, A. Nilsson, INFORMATION SYSTEMS DEVELOPMENT - A Systematic Approach, Prentice-Hall Inc., 1981, pagg. xii + 337

"In this book we summarize nearly ten years of work in the research group called ISAC (Information Systems work and Analysis of Changes) at the Department of Administrative Information Processing at the Royal Institute of Technology and University of Stockholm, Sweden.
Since 1971 the ISAC group has performed research toward a new approach to information systems development. The ISAC work started with the development area information analysis, and gradually grew to include further areas. This growth was largely influenced by practical application in different business companies and other organizations. Today (1981) it covers the development areas' change analysis, activity studies, information analysis, data system design, and

equipment adaptation. The ISAC approach is now in widespread use in the Scandinavian countries in both academic and business sectors. A separate research institute (the Institute for Development of Activities in Organizations) has recently (1981) been founded in Sweden to continue research in the ISAC area. Until now, the ISAC work has been presented in four different Swedish books. Due to international interest, the most comprehensive of these books has now been translated into English, thus the book that you have started to read. [...] This book describes information systems development with special emphasis on analysis and design of information systems. Information systems development consists of analysis, design, and realization (building) of information systems. This book gives an overview of information systems development as well as separate descriptions of each of the four areas of analysis and design of information systems: *activity studies, information analysis, data system design,* and *equipment adaptation.* Before you start an information systems project, you should find out if there really is a need for the development of information systems or if the need is of a different kind. This investigation is called *change analysis.* This book also describes change analysis in a separate chapter. [...]
This book addresses itself to those who either
– Want to acquire an overview picture of the ISAC approach to information systems development or to those who in one way or another have decided that they want to work with the ISAC approach and who
– Want to have examples and guidelines for practical work with the ISAC approach.
This book addresses itself both to those who are using information systems in their daily activities (users) and to those who analyze and design such systems for and together with others (systems analysts and systems designers). This book requires no special previous knowledge except for Chapters 6 and 7. These chapters treat parts of information systems development with data technical elements. In order to understand certain data technical design steps fully, it is therefore useful to have a basic knowledge about computers and their architecture. The links between the different design steps are however possible to understand independent of such background. It is precisely the relations between the different work steps that are essential in order to obtain a total picture of the methods chain of the approach to information systems development."

Indice: 1. Introduction; 2. Overview; 3. Change Analysis (C); 4. Activity Studies (AS); 5. Information Analysis (IA); 6. Data System Design (DSD); 7. Equipment Adaptation (EA); 8. Information Systems Development and People; 9. Epilogue.

Linguaggi di programmazione

● J. Larmouth, FORTRAN 77 PORTABILITY, Software - Practice and Experience, vol. 11 (ottobre 1981), 1071-1117

"This paper is the result of a study of the portability problems which might arise in the use of the Fortran 77 language. The study involved both an examination of the text of the Standard and discussions with a number of compiler-writing teams. The paper identifies areas of Fortran 77 which are likely to cause problems, either because of an incomplete language specification, difficulties in compilation, or user inattention. It will be of interest not only to users writing large packages in Fortran, and to Fortran compiler writers, but also to those involved in language standardization and portability in general."

● D.V. Moffat, 1981 PASCAL BIBLIOGRAPHY (JUNE, 1981), SIGPLAN Notices (ACM), vol. 16, n. 11 (novembre 1981), 7-21

"This bibliography contains references to articles in professional journals and conference proceedings, and to texts and monographs. It does not reference articles in "hobby" magazines (such as BYTE) or in trade journals (such as Datamation). It is a corrected, edited, and updated version of the June, 1980, bibliography. It contains approximately 30% more entries than the previous version. The references are divided by general subject matter into several categories: Language definition and standard; Texts and teaching; UCSD Pascal; Concurrent Pascal; Programming style; Implementation notes and issues; Language comparisons; Discussion and debate. [...]"

Documentazione del software

● R.E. Horn, J.N. Kelly, STRUCTURED WRITING - AN APPROACH TO THE DOCUMENTATION OF COMPUTER SOFTWARE, ACM SIGDOC Newsletter, Asterisk, vol. 7, n. 4 (luglio 1981), 29-33

"This paper will briefly examine three topics: a) the current and most significant problems of computer documentation; b) structured writing and its relationship to conventional prose writing; c) the results of using structured writing for documentation and related technical writing. [...]"

Asterisk è un bollettino dello Special Interest Group on Documentation dell'ACM, e si occupa esclusivamente dei vari aspetti della documentazione tecnica (in particolare, del software).
Sullo stesso numero, la breve nota:

● R.J. Brockmann, TEACHING COMPUTER DOCUMENTATION IN AN ACADEMIC SETTING, ACM SIGDOC Newsletter, Asterisk, vol. 7, n. 4 (luglio 1981), 29-23

contiene una concisa bibliografia sulla documentazione del software (aspetti di "technical writing").

Debugging

● M.S. Johnson, A SOFTWARE DEBUGGING GLOSSARY, SIGPLAN Notices (ACM), vol. 17, n. 2 (febbraio 1982), 53-70

"This glossary contains 291 definitions of terms dealing with the debugging of computer software. The list includes numerous synonyms, as well as the proper names of debugging systems described in the open literature. Terms and definitions have been obtained from various sources: the software-engineering literature, other software-engineering glossaries, and individual contributions."

Manutenzione del software

● B.P. Lientz, E.B. Swanson, PROBLEMS IN APPLICATION SOFTWARE MAINTENANCE, Communications of the ACM, vol. 24, n. 11 (novembre 1981), 763-769

"The problems of application software maintenance in 487 data processing organizations were surveyed. Factor analysis resulted in the identification of six problem factors: user knowledge, programmer effectiveness, product quality, programmer time availability, machine requirements, and system reliability. User knowledge accounted for about 60 percent of the common problem variance, providing new evidence of the importance of the user relationship for system success or failure. Problems of programmer effectiveness and product quality were greater for older and larger systems and where more effort was spent in corrective maintenance. Larger scale data processing environments were significantly associated with greater problems of programmer effectiveness, but with no other problem factor. Product quality was seen as a lesser problem when certain productivity techniques were used in development."

Un libro sulla sicurezza nei sistemi informativi

● U. Bussolati, G. Martella, C. Petronio, LA SICUREZZA NEI SISTEMI INFORMATIVI, Collana di Informatica, Franco Angeli Editore, Milano, 1981, pagg. 319

"Se gli anni '70 hanno registrato la nascita, la crescita, l'affermarsi e l'estendersi delle banche dei dati e dei sistemi informativi nei più svariati settori applicativi, gli anni '80 vedranno lo sviluppo dei problemi legati alla sicurezza e alla protezione di tali sistemi.
Uno dei più gravosi ostacoli alla soluzione del problema della sicurezza è costituito in Italia dalla non immediata reperibilità di specialisti dotati sia di una formazione tecnica adeguata sia di metodologie di pianificazione e di tecniche di progettazione degli interventi.
Questo libro vuole introdurre il lettore al tipo di problemi che il progettista della sicurezza incontra e dare un quadro delle tecniche e delle procedure con cui le possibili soluzioni possono essere trovate.
Dalle tecniche d'identificazione a quelle di controllo dell'accesso ai dati, dall'accuratezza all'integrità degli archivi, dalla critografia all'affidabilità dei dati e del sistema, dai controlli di sicurezza di tipo amministrativo alla protezione fisica degli impianti d'elaborazione, il tutto viene esaminato alla luce della tecnologia attuale e della potenzialità futura.
La presentazione degli argomenti si basa sui problemi e sulle applicazioni che caratterizzano il progetto della sicurezza nei sistemi informativi, dando così più che un'impostazione teorica, un approccio al problema di tipo ingegneristico. Si ritiene infatti che è sempre opportuno non rimandare, specie di fronte a problemi reali, l'incontro tra teoria e pratica o tra esigenze da soddisfare e tecnologie disponibili.
Il testo si rivolge sia agli studenti universitari dei corsi specialistici, sia agli analisti ed ai progettisti delle banche di dati e dei sistemi informativi. Esso tuttavia può costituire un'efficace lettura anche per gli amministratori ed i dirigenti aziendali cui spetti un compito decisionale in relazione ai progetti riguardanti la sicurezza". (copertina)

Indice: 1. Concetti introduttivi; 2. Tecniche per il controllo dell'accesso ai dati; 3. Crittografia; 4. Accuratezza; 5. Affidabilità; 6. Integrità dei dati; 7. Sicurezza fisica; 8. Controlli amministrativi; 9. Tecniche per il controllo della sicurezza (auditing); 10. Legislazione.