# UN ESEMPIO DI MULTIPROGRAMMAZIONE STRUTTURATA. P.S.L.: UN LABORATORIO SOFTWARE

P. Ancilotti - N. Lijtmaer

*Istituto di Elaborazione della Informazione del Consiglio Nazionale delle Ricerche. Pisa*

## 1. INTRODUZIONE

Lo scopo che ci siamo prefissi col progetto del *PSL* (*Pisa Software Laboratory*), che è in fase di realizzazione presso l'Istituto Elaborazione della Informazione, è quello di mettere a disposizione di ricercatori, di progettisti e di studenti uno strumento con il quale essi possano sperimentare la costruzione di nuovi sistemi software. Compito del *PSL* è infatti quello di creare un contesto sperimentale, da cui il nome di laboratorio, nel quale sia possibile svolgere in forma interattiva col sistema, delle esperienze di progetto strutturato di sistemi software modulari [2].

In questo ambito *un sistema software* viene definito come un insieme di moduli indipendenti interconnessi e tali, nel loro insieme, da rispondere alle esigenze dell'utente. Ogni *modulo* è un'unità funzionale del sistema software, ed ognuno di essi è programmato indipendentemente dagli altri, utilizzando le regole e i costrutti linguistici della programmazione strutturata: selezione, concatenazione e ripetizione. Ogni modulo viene programmato come un processo ciclico. I vari moduli sono connessi tra loro mediante dei *collegamenti standardizzati*, forniti dal meccanismo di comunicazione del *PSL*, attraverso i quali si scambiano messaggi [1]. Durante l'esecuzione del sistema ogni singolo modulo viene trattato dal *PSL* come un *processo sequenziale*.

Ricapitolando un sistema software può essere considerato: *a*) da un punto di vista *statico* come un insieme di *moduli indipendenti* e *b*) da un punto di vista *dinamico* come una *famiglia di processi asincroni cooperanti*.

Dopo una breve analisi dei concetti di affidabilità, modularità e concorrenza spiegheremo il *PSL*.

Il progetto del *PSL* è un *tentativo* di fornire un sistema interattivo che permetta la costruzione di sistemi software multiprogrammati, strutturati e modulari con un alto grado di affidabilità.

## 2. AFFIDABILITÀ

Con affidabilità di un sistema s'intende normalmente, l'abilità del sistema stesso a svolgere correttamente le sue funzioni indipendentemente da eventuali guasti nei suoi compo-

nenti, dove con guasto di un componente indichiamo un cambiamento, permanente o temporaneo, delle sue caratteristiche e tale da modificare le sue funzioni [4].

Un tale concetto di affidabilità viene definito anche nel campo del software, salvo il fatto che viene dato un *diverso significato al termine guasto di un componente*. Infatti, in questo caso con guasto d'un componente software va inteso ogni tipo di malfunzionamento del componente stesso derivante da un errore commesso durante la fase di progetto o di codifica del componente e che non è stato rilevato nella fase di collaudo o « debugging ». In questo caso il guasto d'un componente è cioè, un qualcosa strettamente correlato alla non correttezza del componente stesso.

All'aumentare delle dimensioni d'un sistema software e della complessità delle funzioni che lo stesso deve svolgere aumenta enormemente la probabilità d'inserire errori nei componenti del sistema e conseguentemente diminuisce la sua affidabilità.

In generale, dato un sistema costituito da $n$ componenti e detta $p$, la probabilità che il generico componente sia corretto, la probabilità $P$, che l'intero sistema sia corretto è dato da:

$$P = p^n$$

Se vogliamo che $P$ sia sensibilmente diversa da zero, dovremo avere una probabilità $p$ per il generico componente quasi uguale ad 1, e questo è tanto più vero quanto maggiore è il numero $n$ dei componenti del sistema, cioè quanto più esso è complesso.

Per aumentare, quindi, l'affidabilità di un complesso sistema software è necessario *seguire certe metodologie di progetto* atte a semplificare una sua eventuale *verifica di correttezza*, a rilevare il maggior numero possibile di errori in *fase di compilazione*, e a facilitare la fase *di « debugging » e documentazione*.

L'affidabilità di un sistema software sotto *PSL* si basa sulla *modularità* e sull'applicazione delle *metodologie della programmazione strutturata*.

## 3. MODULARITÀ

Una delle caratteristiche più importanti di un sistema software è rappresentata dalla facilità con cui lo stesso può essere *modificato o ampliato* come conseguenza di variazioni del contesto in cui il sistema deve operare. Tali variazioni possono consistere nella modifica di alcune funzioni che il sistema deve svolgere o nella modifica in certe strutture di dati che il sistema deve elaborare o addirittura in una variazione, nel numero e nel tipo, di risorse fisiche su cui i sistema opera.

La modularità della programmazione è una delle tecniche usate per raggiungere lo scopo nel progetto di sistemi software modificabili ed estendibili [8].

Informalmente un sistema software, come qualunque altro sistema, è detto modulare se è possibile suddividerlo in un certo numero di componenti principali, ognuno di questi componenti è a sua volta divisibile nei propri sotto componenti e così via.

Nel caso di sistemi software, il livello a cui possiamo scendere nella suddivisione del

sistema è connesso con i meccanismi di cui disponiamo per combinare moduli di programmi in programmi più grandi. Fra i fattori più importanti di cui è necessario tener conto vi sono:

*a*) La possibilità di interdipendenze tra un modulo di programma ed un altro;
*b*) La facilità con cui tali interdipendenze sono eliminate;
*c*) L'istante in cui sono eliminate.

Nel caso in cui i moduli di programma siano entità completamente indipendenti l'uno dall'altro, è possibile usarli come blocchi componenti che possono essere arbitrariamente combinati per formare sistemi complessi. Ciò conduce al concetto di *sistema di programmazione* visto come un *insieme di moduli di programma* da cui l'utente può prelevare i componenti che, opportunamente collegati, costituiscano un sistema atto a soddisfare le sue esigenze.

Nella letteratura [5], si trovano vari richiami relativi alla programmazione modulare, i quali, pur essendo formalmente diversi, hanno in comune il principio di « *divede et impera* » e lo scopo di ridurre l'alto costo di produzione di sistemi software correttamente funzionanti.

Dennis [5] definisce la *modularità come una proprietà del sistema di elaborazione* e non come una proprietà che può appartenere o meno ad un programma. Inoltre stabilisce due obbiettivi a cui la programmazione modulare deve tendere:

*a*) Un programmatore deve convincersi della correttezza di un modulo di programma indipendentemente dal contesto in cui verrà usato quale componente di programmi più complessi;
*b*) Deve essere possibile collegare moduli di programma, scritti da programmatori diversi, senza la necessità di conoscere la logica interna d'ogni modulo ma solo le sue funzioni.

Un sistema di elaborazione possiede la proprietà della modularità solo se il *livello linguistico* definito dal sistema gode delle precedenti caratteristiche, dove con livello linguistico s'intende il linguaggio in termini del quale viene scritto il « software » per il sistema stesso. È cioè necessario che al livello linguistico sia associata:

*a*) Una *classe di oggetti* che sono le rappresentazioni delle unità di programma o moduli;
*b*) Un *meccanismo per combinare moduli* tra loro che abbia come scopo la costruzione di moduli più complessi senza modificare minimamente i moduli componenti.

Al concetto di modulo di programma vengono dati due significati diversi. Da una parte viene associato il concetto di *modulo* con il concetto di *procedura*. In questo senso un modulo può richiedere, in ogni istante, l'attivazione di un'altra procedura. Il collegamento fra i moduli è realizzato mediante un normale passaggio di parametri, sia per i dati iniziali che per i risultati. In questo contesto un programma modulare è *un insieme di procedure non interferenti*. Vedremo, però, che la procedura *non* soddisfa completamente i due obbiettivi della programmazione modulare. Un secondo significato che viene dato al concetto di modulo è quello di *un'entità funzionale indipendente* che può essere collegata ad altri moduli per mezzo di un *insieme di connessioni*. Ogni modulo riceve dati dalle sue connessioni d'ingresso,

elabora questi dati, e invia i risultati alle connessioni di uscita. Queste connessioni, attraverso le quali i moduli vengono collegati, possono essere viste come dei canali attraverso cui avviene lo scambio d'informazione. Generalmente vengono inserite delle code nei collegamenti per aumentare l'efficienza del meccanismo di scambio di messaggi («*Mailboxes*») [2].

In questo secondo tipo di modularità, che come vedremo soddisfa perfettamente gli obiettivi della programmazione modulare, ogni modulo è continuamente *attivo*, elaborando dati, finchè vi sono dati d'ingresso, e, in contrapposizione al concetto di procedura, più moduli possono essere attivi contemporaneamente, cioè, *la concorrenza* è una caratteristica propria di questo secondo schema [7]. Il concetto di modulo è in questo caso strettamente connesso al concetto di « *coroutine* » introdotto da Conway [3] e al concetto di *processo sequenziale* [6, 10].

Vediamo adesso, brevemente, come le *procedure* e/o i *sottoprogrammi* introdotti da linguaggi di programmazione noti (FORTRAN, ALGOL) creano delle limitazioni alla programmazione modulare [5].

### 3.1. Modularità nei linguaggi di programmazione

Un *programma* FORTRAN consiste di un programma principale (« MAIN ») ed eventualmente di un insieme di sottoprogrammi di tipo « FUNCTION » o « SUBROUTINE ». Nel FORTRAN non è possibile combinare tra loro due programmi scritti separatamente, è quindi evidente che un programma principale non può costituire un modulo. Consideriamo quindi, il sottoprogramma come un modulo. La prima limitazione deriva precisamente, dal fatto che più sottoprogrammi per esser collegati hanno bisogno di un programma principale. *La possibilità di creare moduli più complessi mediante combinazione di moduli si ferma, dunque, al primo livello.*

Altre limitazioni sorgono dal fatto che due sottoprogrammi possono essere stati chiamati con lo stesso nome e quindi originare dei conflitti se usati insieme. Un altro conflitto sui nomi può sorgere con il « LABELLED COMMON ». Inoltre due sottoprogrammi possono usare il « BLANK COMMON » per scopi diversi producendo interferenze non desiderate se combinati arbitrariamente.

Tutto ciò può condurre alla necessità di *alterare la rappresentazione di un modulo* prima di poterlo combinare con altri e ciò è in *contrapposizione* con gli obbiettivi della programmazione modulare.

Consideriamo adesso la *procedura « ALGOL »* quale modulo di programma. Le procedure possono essere combinate per formare procedure più complesse a differenza dei sottoprogrammi del « FORTRAN ». In questo caso un programma modulare consiste in una *gerarchia di procedure*. È necessario però che un modulo sia rappresentato da una procedura in linguaggio sorgente in quanto programmi « ALGOL » compilati non rappresentano più moduli del livello linguistico « ALGOL » e come tali non possono essere più combinati.

Un secondo inconveniente deriva dal meccanismo dei riferimenti a *variabili non locali* dell'« ALGOL ».

Per es.: data la procedura

*integer procedure* $P(x, y)$; *integer* $x, y$;
*begin*
$P := x + y * i$ ;
$i := i + x$ ;
*end*

In questo caso l'identificatore $i$ rappresenta una variabile non locale alla procedura $P(x, y)$. Per usare la procedura come modulo è quindi necessario conoscere tutti i riferimenti a variabili globali che vi sono in essa e nelle eventuali procedure in essa racchiuse, in quanto tali riferimenti non locali costituiscono un mezzo d'interazione tra procedure.

Se per esempio due procedure $P$ e $Q$ usano lo stesso identificatore globale $i$ e se esse sono racchiuse all'interno di una stessa procedura più esterna, sorge un conflitto fra i nomi. Quindi l'uso di *riferimenti globali* in ALGOL *viola* gli obiettivi della modularità.

Altri commenti sul PL/I e sul LISP si possono trovare, nel lavoro di Dennis [5].

## 4. CONCORRENZA

I concetti di concorrenza e di funzionamento asincrono sono di fondamentale importanza nell'analisi e nel progetto di complessi sistemi software, in particolare quando, secondo la definizione di modularità, ognuno di essi sia costituito da un insieme di moduli indipendenti interagenti attraverso particolari connessioni stabilite dal progettista di ogni modulo.

Se due moduli $A$ e $B$ di un sistema sono progettati indipendentemente, il verificarsi di un evento $a$ nel modulo $A$ può essere vincolato al verificarsi di un evento $b$ nel modulo $B$ esclusivamente come conseguenza di un'interazione fra i due moduli $A$ e $B$. Ciò significa che, fintanto che non ci sono interazioni fra i due moduli, il verificarsi di eventi in ognuno di essi, e cioè la loro evoluzione, può procedere concorrentemente senza che *nessuna relazione temporale* sia necessariamente definita fra di loro. L'imporre una relazione temporale tra il verificarsi di eventi indipendenti in moduli distinti, significa porre dei vincoli non necessari che complicano inutilmente il sistema, rendendolo più difficile da analizzare a partire dal testo e da modificare. Inoltre introducono dei ritardi inutili che possono ridurne l'efficienza.

È quindi naturale che l'insieme dei moduli costituente il sistema software sia trattato, durante l'esecuzione, come una *famiglia di processi sequenziali asincroni* [6, 10].

Come risultato il progettista è incoraggiato a *decomporre il sistema* in parti concorrenti d'accordo a quella che è la logica interna del sistema stesso, sfruttando inoltre vantaggiosamente il parallelismo proprio dello « hardware » del sistema di elaborazione.

Poichè, durante l'esecuzione, ogni modulo da luogo ad un processo sequenziale che evolve in modo asincrono rispetto agli altri e poichè nel loro complesso i vari processi debbono cooperare per il raggiungimento di un unico scopo, è necessario che il livello linguistico in cui sono scritti i vari moduli, fornisca un *insieme di primitive* mediante le quali sia possibile *sincronizzare* le attività dei processi nei punti in cui essi debbono interagire [1].

L'interazione tra i processi consiste, come abbiamo già accennato, in uno *scambio di messaggi*; è pertanto opportuno che le primitive di sincronizzazione siano le più idonee a risolvere tale tipo di interazione in modo *ben strutturato*.

## 5. PROCESSI CONCORRENTI NEL PSL

Il progetto del PSL è stato sviluppato tenendo presente l'obiettivo primario di generare un contesto sperimentale per la stesura di sistemi software con un alto grado di affidabilità. È stato già accennato come, per questo motivo, abbiamo ritenuto opportuno ricorrere alle tecniche della programmazione strutturata e della programmazione modulare e come il concetto di modularità, a cui ci siamo riferiti, sia strettamente connesso con il concetto di concorrenza. Il primo aspetto che abbiamo preso in considerazione nel progetto del PSL, è stato appunto quello di creare un sistema che godesse della proprietà della modularità, proprietà, che come abbiamo visto, deve appartenere al sistema di elaborazione.

Ricordiamo (par. 3) che, perchè un sistema goda della modularità, è necessario che al livello linguistico siano associate due condizioni. Per soddisfarle abbiamo separato in due classi le funzioni del PSL. Nella prima classe abbiamo inserito tutte le funzioni preposte alla *scrittura, caricamento e collegamento dei moduli*, funzioni che sono svolte in fase statica dal PSL, prima cioè che il sistema software passi in esecuzione. Nella seconda classe vi sono tutte le funzioni preposte al *controllo dinamico* delle attività dei processi, cioè: *simulazione dei processi, comunicazione fra i processi e controllo delle risorse fisiche.*

L'insieme delle due classi di funzioni che il PSL deve fornire costituisce il *nucleo del PSL*, suddiviso, appunto, in due parti gerarchicamente organizzate; il *nucleo dinamico* al livello più interno e il *nucleo statico* a livello più esterno.

### 5.1. Nucleo dinamico

Il nucleo dinamico del PSL è un'estensione « software » dello « hardware » del sistema, nel nostro caso l'IBM 360/65.
Gli scopi del nucleo dinamico sono:

*a*) gestire le attività dei singoli processi, generando un « processor » virtuale per ogni processo e garantendo la loro completa indipendenza;

*b*) creare un insieme di primitive mediante le quali sia possibile realizzare tutte le funzioni necessarie ai processi per svolgere una corretta cooperazione;

*c*) realizzare mediante le precedenti primitive, un insieme di funzioni che siano le più idonee per ben strutturare i due tipi di interazione che possono verificarsi tra i processi: interazioni *dirette*, per lo scambio di messaggi e interazioni *indirette* per la competizione sull'uso delle risorse fisiche [2].

Il *nucleo dinamico* è stato a sua volta realizzato mediante una *struttura gerarchica* costituita da *tre livelli* che hanno il compito di ottenere, rispettivamente, i tre scopi sopraenunciati.

Il primo livello è preposto, quindi, al compito di ripartire l'uso dell'unità centrale assegnandola ad ogni processo per un preciso intervallo temporale (« time-slice ») prima di assegnarla ad un altro. A questo livello viene opportunamente sfruttato il meccanismo di protezione fornito dal sistema 360 in modo di realizzare la completa indipendenza di un processo da tutti gli altri. Come conseguenza di ciò ogni modulo di programma ha un *proprio spazio dei nomi* e può fare riferimento solo alle *proprie variabili locali*. Ciò offre due vantaggi:

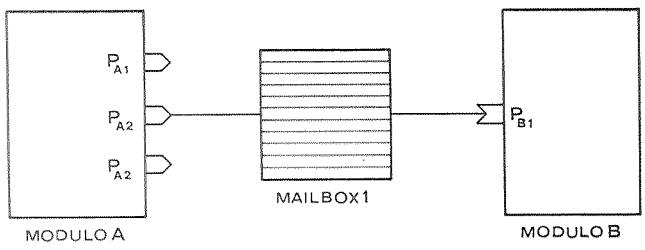
*a*) il grado di modularità di cui gode un sistema software, sotto PSL, risponde perfettamente agli obiettivi della programmazione modulare, *senza restrizioni*;

*b*) poichè un sistema software, sotto PSL, è modulare, ed ogni modulo è un processo sequenziale autonomo, è possibile programmare ogni modulo applicando le tecniche della programmazione strutturata.

Al secondo livello vengono realizzate le primitive di sincronizzazione: $P$ e $V$ che operano su variabili di tipo semaforo. Mentre con il primo livello abbiamo creato un insieme di « processors » virtuali, uno per ogni processo, del tutto equivalenti al « processor » fisico, con l'aggiunta del secondo livello abbiamo semplicemente incrementato la capacità logica di ogni « processor » virtuale fornendogli la capacità di eseguire le primitive $P$ e $V$. Queste « primitive » però non sono disponibili al di fuori del nucleo ed è invece al terzo livello che vengono realizzate, sfruttando opportunamente le precedenti primitive, le funzioni più idonee per lo scambio di messaggi [1], e per la gestione delle risorse fisiche.

### 5.1.1. *Meccanismo per lo scambio di messaggi*

Per realizzare un efficace meccanismo di scambio di messaggi tra processi, ogni modulo ha un certo numero di *porte* attraverso le quali questo può ricevere oppure inviare informazione sotto forma di messaggi. L'unica possibilità che il modulo ha per scambiare informazione, è attraverso una delle sue porte. Affinchè due moduli $A$ e $B$ possano comunicare tra di loro è necessario che una porta di uscita di $A$ sia connessa ad una porta d'ingresso di $B$. Questo collegamento viene effettuato mediante un « buffer » di messaggi, detto « *mailbox* » e lo scambio d'informazione avviene secondo il noto schema del produttore-consumatore [1]. Il collegamento fra moduli, *sempre unidirezionale*, consiste quindi nella sequenza *porta-mailbox-porta*



Una volta che due moduli sono così collegati la porta coincide con il nome, locale ad ogni modulo, del collegamento. Attraverso una mailbox possono, però, collegarsi *più di due moduli*.

Le funzioni che sono state realizzate per lo scambio di messaggi sono:

*SEND* ($m$, $p$), dove $m$ è il nome del messaggio e $p$ è la porta attraverso la quale viene inviato. Tale funzione controlla, per prima cosa, se la « mailbox » collegata alla porta $p$ può ricevere il messaggio oppure se è piena. Nel primo caso il messaggio $m$ viene inserito opportunamente nella « mailbox ». Se invece quest'ultima è piena, il processo che ha invocato l'esecuzione della *send* viene *bloccato* per essere riattivato quando nella « mailbox » vi sarà di nuovo spazio per il messaggio.

*RECEIVE* ($p$), dove $p$ è il nome della porta attraverso la quale si vuole ricevere il messaggio. Se la « mailbox » connessa con la porta $p$ contiene almeno un messaggio, l'informazione viene resa disponibile al processo che ha invocato l'esecuzione della *receive*. Se invece la « mailbox » è vuota il processo viene *bloccato* per essere *riattivato* quando qualche messaggio sarà stato inserito nella « mailbox ».

Questo meccanismo per lo scambio di messaggi ha la caratteristica che ogni modulo *conosce solo* il *nome delle proprie porte* e non sa quale altro modulo vi è collegato. Ciò coincide con l'aver *eliminato* tutti i nomi di *variabili globali* all'interno dei moduli, che quindi possono esser interconnessi in modo arbitrario senza dover modificare minimamente i moduli stessi.

### 5.1.2. *Meccanismo per la gestione delle risorse fisiche*

Per quanto riguarda il controllo dei *dispositivi d'ingresso-uscita*, abbiamo realizzato un'*unica funzione*, chiamata *DO I/O* [9], che ha come scopo l'*eliminazione degli effetti collaterali delle interruzioni automatiche*. Infatti abbiamo cercato di evitare che i segnali d'interruzione si manifestino al programmatore, pur continuando ad esistere come mezzo « hardware » di sincronizzazione tra unità centrale e dispositivi periferici. L'inconveniente più importante introdotto dalla interruzione automatica è dovuto al fatto che un insieme d'istruzioni viene inserito in punti del programma che non possono essere previsti nè ricostruiti. Da ciò ne deriva che il programma che viene eseguito *non può essere direttamente controllato sul testo scritto e non può essere garantita la riproducibilità dei risultati* [1]. Per evitare questo inconveniente, un processo che ha eseguito una *DO I/O* resta bloccato fino a quando non arriva il corrispondente segnale di terminazione dell'operazione d'ingresso-uscita e viene attivato un altro processo. Ciò è in accordo con il principio di stretta sequenzialità all'interno di ogni processo e inoltre incoraggia il programmatore a suddividere il sistema in parti concorrenti, secondo quella che è la logica naturale del sistema stesso.

Sono state realizzate, infine, due funzioni: *GET e RELEASE* mediante le quali ogni processo può richiedere o rilasciare *dinamicamente* spazio in memoria centrale.

### 5.2. Nucleo statico

È il livello più esterno del nucleo ed è preposto al controllo delle *interazioni tra l'utente e il nucleo dinamico* durante la fase statica di caricamento e interconnessione dei moduli. Per questo scopo viene fornito all'utente un particolare sistema software *estendibile da parte dell'utente stesso*, chiamato *linguaggio di controllo*, che accetta una serie di comandi da terminale e li esegue sfruttando le funzioni del nucleo statico. Durante la fase di creazione del sistema software, l'utente, per mezzo dei comandi del linguaggio di controllo, carica in memoria i singoli moduli e li interconnette realizzando così la struttura topologica del suo sistema. Quindi fissa certi parametri, modificando i quali può rilevare sperimentalmente alcune caratteristiche del sistema. Tali parametri sono: le *priorità d'ogni modulo*, di cui si tiene conto nell'algoritmo di « scheduling », le *dimensioni delle varie « mailboxes »*, e l'ampiezza del *quanto di tempo*.

Terminata la fase statica, per mezzo del comando di *START*, il sistema software, passa in esecuzione.

## 6. PROGETTO DI UN SISTEMA SOFTWARE

Il progetto di un sistema software nel *PSL* può essere logicamente diviso in due fasi distinte. Nella prima fase il sistema da realizzare viene analizzato in modo da determinarne la struttura tenendo conto delle funzioni che deve svolgere. In questa fase vengono quindi ricavate le specifiche dei singoli moduli componenti e delle connessioni fra i vari moduli.

Nella seconda fase si procede alla programmazione di ogni modulo. Tale fase può essere parzialmente, o del tutto eliminata qualora si disponga già dei moduli necessari.

Terminate queste due fasi di progetto si passa direttamente alla realizzazione topologica del sistema sul *PSL*, mediante il linguaggio di controllo, e quindi alla sua esecuzione. Un esempio di progetto per un sistema di aggiornamento di « files » si trova in [2].

## 7. CONCLUSIONI

All'aumentare delle dimensioni e soprattutto della complessità d'un sistema « software » aumenta enormemente la probabilità di fare errori durante la fase di progetto.

D'altra parte la fase di collaudo, « debugging », di un sistema diventa molto costosa. Come giustamente fa osservare Dijkstra, tale fase ci permette di rilevare certi errori, ma non ci assicura della loro completa eliminazione. È quindi necessario che siano seguite certe regole durante la fase di progetto del sistema sia per diminuirne la complessità logica, sia per semplificare la fase di collaudo rilevando la maggior parte degli errori in fase di compilazione.

Le regole che abbiamo seguito nel progetto del *PSL* sono state appunto, quelle di fornire all'utente un contesto sperimentale nel quale sia possibile applicare, con una certa facilità, le tecniche della programmazione strutturata e quelle della programmazione modulare in modo tale da ottenere sistemi « software » affidabili, modificabili ed estendibili.

## 8. BIBLIOGRAFIA

[1] P. Ancilotti, M. Boari, N. Lijtmaer, *Tecniche di Programmazione Strutturata Estese ad un Ambito di Processi Concorrenti*. Seminario su Programmazione Strutturata, Milano 12-13 Febbraio 1974.

[2] P. Ancilotti, R. Cavina, M. Fusani, F. Gramaglia, N. Lijtmaer, E. Martinelli, C. Thanos, *Designing a Software Laboratory*, Proceedings of the 8-- Yugoslav International Symposium, Bled 1973.

[3] M. E. Conway, *Design of a Separable Transition-Diagram Compiler*. Comm. ACM. 6. 7 1963.

[4] J. B. Dennis, *The Design and Construction of Software Systems*. Advanced Course on Software Engineering F. L. Bauer, editor Springer-Verlag, 1973.

[5] J. B. Dennis, *Modularity*. Advanced Course on Software Engineering. F. L. Bauer, editor, Springer-Verlag, 1973.

[6] E. W. Dijkstra, *Cooperating Sequential Processes*, in Programming Languages, F. Genuys, ed. Academic Press, 1968.

[7] E. Morenoff and J. B. McLean, *Inter-Program Communications, Program String Structures and Buffer Files*. Proceedings AFIPS Spring Joint Computer Conference 1967.

[8] *Nato Report* on *Software Engineering* Rome, Italy October 1969.

[9] N. Wirth, *On Multiprogramming, Machine Coding, and Computer Organization*. Comm. ACM 12, 9, 1963.

[10] J. B. Dennis, *Cotoutines and Parallel Computations*. Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems, 1971.