C84-18

RACCOLTA DELLE TRASPARENZE PER LAVAGNA LUMINOSA USATE DURANTE IL CORSO "I LINGUAGGI DI SPECIFICA PER LA PRODUZIONE DI SOFTWARE" SVOLTO, PRESSO IL CNUCE DI PISA, NEL PERIODO 5-7 NOVEMBRE 1984.

La raccolta delle trasparenze usate durante il Corso sui Linguaggi di Specifica per la produzione di Software e', anche se un po' in ritardo, pronta.

Spero che questa raccolta possa essere utile ai partecipanti al Corso per poter ripercorrere i vari interventi e rimettere a fuoco almeno i punti centrali sviluppati dai docenti.

Nel distribuire questo fascicolo colgo nuovamente l'occasione per ringraziare tutti i docenti del corso che con la loro qualificata presenza hanno determinato il successo di questa iniziativa.

Un grazie particolare va al Prof. Ugo Montanari ed al Dott. Marco Bellia per l'impostazione e la discussione circa i contenuti del Corso stesso ed alla Sig.ra Elena Lofrese per aver risolto i vari problemi organizzativi.

Maurizio Martelli

Lista dei docenti

Prof. Antonio Albano
Dipartimento di Informatica - PISA

Prof. Egidio Astesiano
Universita' di Genova

Dott. Roberto Barbuti
Dipartimento di Informatica - PISA

Dott. Marco Bellia
Dipartimento di Informatica - PISA

D.ssa Linda Crimersmois
Italsiel - ROMA

Dott. Pierpaolo Degano
Dipartimento di Informatica - PISA

Prof. Marco Malocchi
Ist. di Cibernetica - Universita' di Milano

Dott. Maurizio Martelli
CNUCE - PISA

Prof. Ugo Montanari
Dipartimento di Informatica - PISA

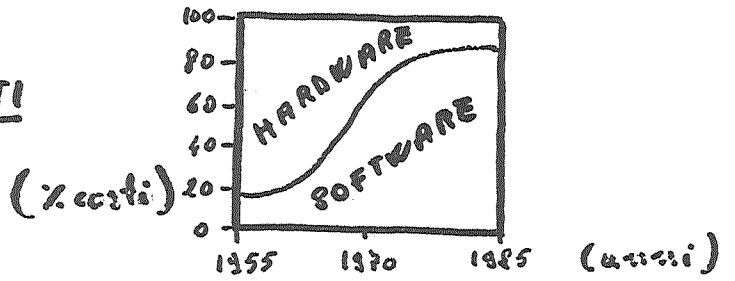
Prof. Knut Ripken
Laboratoires de Marcoussis (F)

INTRODUZIONE: INGEGNERIA DEL SOFTWARE

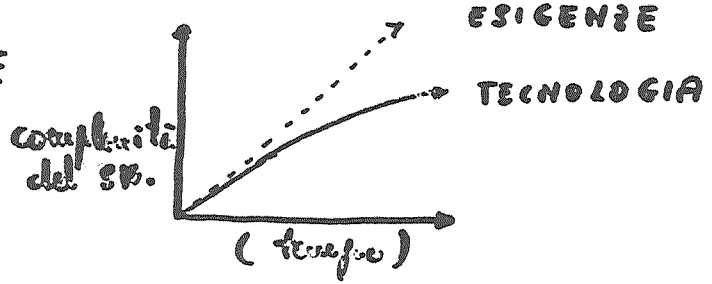
tratto dalle "Lezioni del software"
Dott. MAURIZIO MARTELLI

TRENDS:

COSTI



ESIGENZE



DEFINIZIONE DI INGEGNERIA DEL SW. (BAUER)

Istituzione ed uso di appropriati principi (metodi) per ottenere economicamente software che sia affidabile e che funzioni su macchine reali.

S.E. Disciplina Ingegneristica
⇒ attenzione alla realizzazione pratica.

S.E. Sviluppi Teorici.
- AUTOMAZIONE DEL PROCESSO DI COSTRUZIONE DEL S.W.
- SVILUPPO TOOLS AVANZATI
⋮

⇒ profonda influenza

LINGUAGGI DI SPECIFICA:
un aspetto essenziale del problema specialmente se visti inseriti negli ambienti di sviluppo e misurati rispetto a tutto il ciclo di vita del software.

Linguaggi di Specifica

Dott. Marco Bellia

- Sistemi software di grandi dimensioni
  - 1 - 2 ordini di grandezza maggiori dei sistemi ordinari (100 righe di codice)
  - sviluppati in un lungo arco di tempo ed impiegando gruppi di analisti / programmatori
  - concettualmente semplici ma coinvolgenti molti fattori (dall'Hardware al tipo di utenza)

- Ciclo di sviluppo Analisi – Programmazione è insufficiente.
  - linguaggi di programmazione non bastano

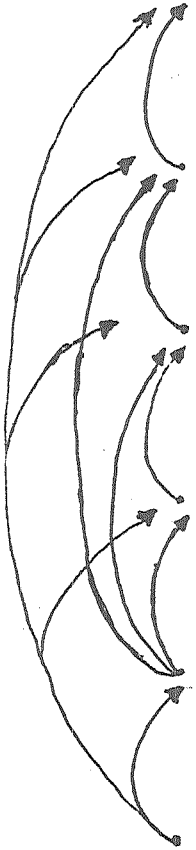
1968 - 1969

- due conferenze sul "Software Engineering" sponsorizzate dalla NATO [1]
- si parla di "software crisis"
- Necessità di "regole": no alla "programming art"

Dalle conferenze emerge:

- Caratteristiche di un "buon software"
- Principi da seguire nello sviluppo di un "buon software"
- Fattori che intervengono nella costruzione di un "buon software"

Tali fattori danno origine ad un CICLO di VITA



Analisi dei requirements (A-R)
"Cosa vuole l'utente"?
interfaccia sistema – ambiente

Specifica (SP)
"Cosa deve fare quel sistema"?
caratteristiche (funzionali) del sistema

Progetto (PR)
"Come deve fare quel sistema"?
piano di sviluppo del sistema

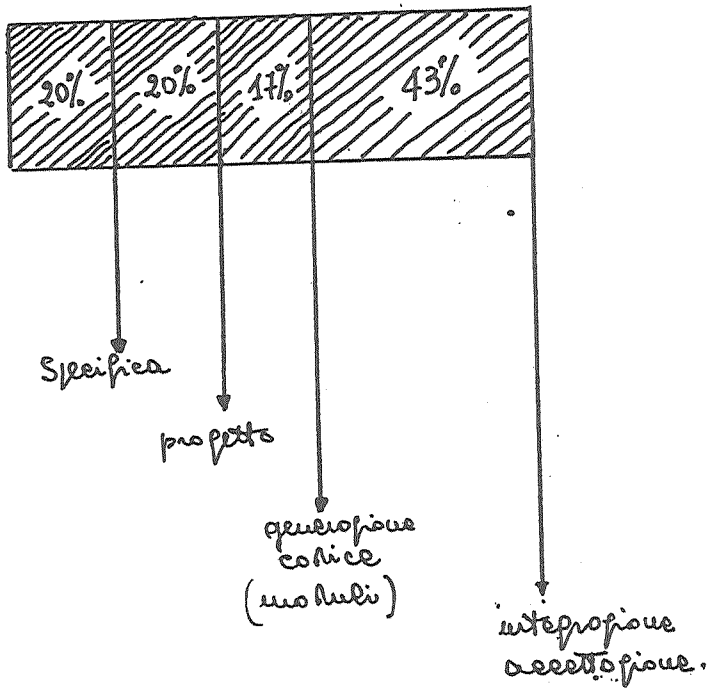
Generazione del codice (CO)
"Come è fatto"?
implementazione / programmazione

Testing (TS)
"Ciò che è fatto è quanto deve fare quel sistema"?
aderenza alle richieste

Uso e Manutenzione (U-M)
"Che sarà del sistema prodotto"?
gestione del sistema prodotto

Il ciclo di vita è dinamico

Distribuzione delle risorse



linguaggi di specifico

linguaggi proposti: considerazioni generali

- Sviluppati nel decennio 1970 - 1980

- Sono tanti [1] (meno dei L.P.)

  - Società

    + grossa società da in quanti anni abbia avuto la necessità di sviluppare suoi sistemi software ha messo a punto un proprio L.S.

  - Software Houses

    Sono state le prime e con la loro esperienza ....

  - Ricerca

    come per tutti i linguaggi, nuove tecniche e/o considerazioni di vecchie hanno suggerito miglioramenti e talvolta nuovi linguaggi +

- Diversamente dai linguaggi di Programmazione

  - Organico insieme di strumenti e di metodologie (di specifica)

  - facilities appositamente studiate per affrontare lo sviluppo di particolari classi di sistemi

- Lo sviluppo del software non è standard

  - No demarcazione tra linguaggi di specifica e linguaggi di progetto

  - linguaggi ad ampio spettro

Linguaggi di Specifica

Linguaggio di specifica ≡ strumento per definire formalmente la specifica

Requisiti di un ling. di spec.

Riferibilità
precisa regole matematiche e semantiche

Non-ambiguità
specifiche espresse in modo preciso ed univoco

Universalità
tutti gli aspetti del problema sono esprimibili

Varietà e disomogeneità degli aspetti da specificare (requirements di sistema) inducono ad un compromesso tra universalità e riferibilità

struttura e funzionalità
+
tipologia utenza (ambiente di uso)
visione target
prototipazione
affidabilità
robustezza
costi e readeguo
vita media sist.

Minimalità
linguaggio non deve fornire l'uso di simboli non propri del sistema.

Integrità
- specifiche proprie
- integrità della estensione

Modificabilità
modifica semplice e rapida

Meccanismi di astrazione e di modularizzazione

Linearità
complessità e dimensioni della specifica linguaggi
con complessità e dimensioni del sistema

Concorrenza
più parti della specifica in conflitto tra loro

- Nessun specifico meccanismo
- Tecniche e metodi del linguaggio (amiomatico)

Implementabilità
- specifiche effettive
- utilizzabili per sviluppare il progetto

Ogni costrutto del linguaggio è caratterizzabile in termini di prestazione calcolata

oppure

Il metodo mancano <u>no</u> espressioni calcolabili
- problemi

Tracciabilità
mostrare il comportamento del sistema specificato.

- Interpreti simbolici
- simulatori
- Interpreti umanici (linguaggi di specifica e prototipazione)

Pragmatica semplice
simile senza specifiche esigenze

Multi - descrittivo
offre molti - letture differenziate per utenza

- meccanismi di modularizzazione che rendono visibili solo parti della specifica a seconda per tipo di lettura.
- strumenti di analisi e comprensione di documentazione
- Tecniche e metodi determinano la pragmatica.

Non sono requisiti di un L.P.

- Concisione: efficiente uso delle regole (sintassi non sempre ben definita)
- Efficienza: regole efficienti (analisi più che ambiti)
- Non-ridondanza: possibilità di ripetere aspetti già descritti (sovraridondanza di specifica)
- Espressività: non è un linguaggio di programmazione

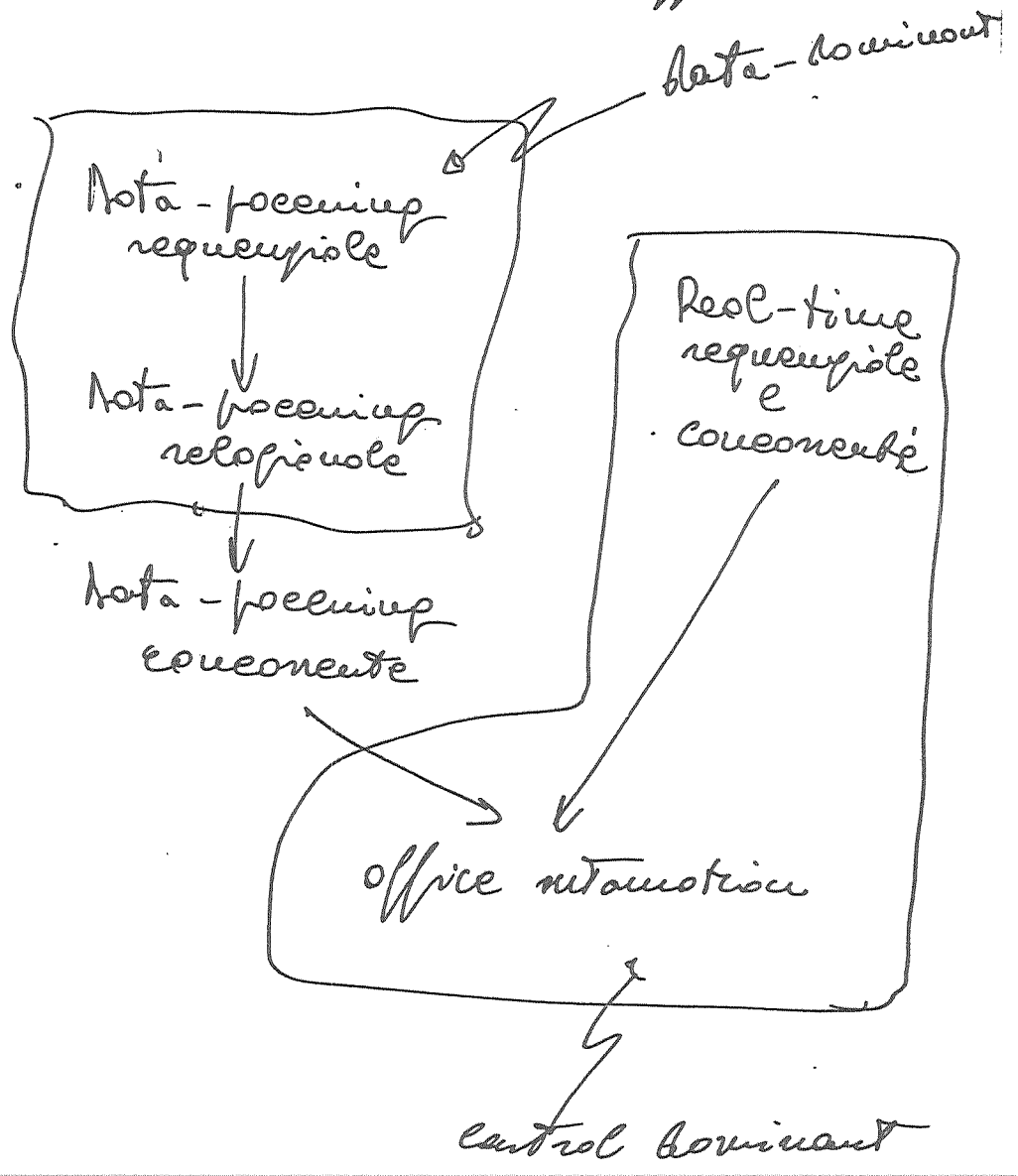
Caratteristiche e Classificazione

- Non esiste una "teoria" di questi linguaggi
- Requisiti rappresentano una classificazione
  - classe dei sistemi sviluppabili
  - Fasi del ciclo considerate
  - Tecnica di specifica
  - Metodo di specifica

Fasi del ciclo di vita classiche

- documentazione requirements
- specifica architetturale
- specifica funzionale
- progetto
- implementazione

Classe di sistemi orientati



Data processing sequenziale

- gestione di grosse quantità di dati
- dati strutturati (poco)
- archivi
- controllo semplice (sequenziale nelle attività)
- funzionalità semplice

es.: sistemi di accounting (paghe, ...)

Data processing relazionale

- gestione di grosse quantità di dati
- dati strutturati (molto)
- archivi
- relazioni che vincolano tra loro struttura e modificabilità di dati
- controllo semplice : - sequenzialità
  - interattività
- funzionalità semplice

es.: data base management systems
sistemi informativi (reservation)

PSL/PSA - ISDOS
OBJ

Data processing concorrente

- grosse quantità di dati
- dati strutturati (poco - molto)
- archivi
- controllo di processi indipendenti e diversi (parallelismo di attività)

Es.: sistemi informativi con gestione distribuita dei dati (sistemi bancari, ...)

SA/SADT - SOFTECH

Real time

- vincoli di tempo
  - quantità dati processabili / unità di tempo
  - sincronizzazione di processi

Es.: sistemi di controllo processi industriali e di arma

RLS/SDS - BKBATC
RLP - Honeywell

requisiti di specifica

(8)

Office automation

- sintesi dei precedenti

ES.: sistemi da impiegare in strutture organizzative allo scopo di arricchire l'intera struttura organizzativa ed di integrare attività umane con attività automatizzate.

MOLTE PROPOSTE

CNUCE

Documentazione Requirements

- Descrizione Requirements

1. descrizione degli obiettivi

2. descrizione del tipo di sistema
   + interfaccia con altri sistemi (hardware, software, umani)
   + metodologie di uso del sistema (uso 1 volta, uso periodicamente, uso come front-end ...).

(3) - descrizione dei constraints del prodotto
   + risorse utilizzabili (hardware-software) del target
   + valori di performance (tempo di processamento, ...
   + valori di affidabilità
   + valori di robustezza
   + prob di fallimento

(4) - descrizione dei constraints nel processo di proj.
   + risorse utilizzabili dell' host
   + costi e tempi
   + modalità di check nello sviluppo del prodotto

- specifica architetturale

funzionalità globale descritta dai requirements

⇓

decomposizione del sistema in attività principali:

- definizione funzionalità dei componenti
- definizione input e output dei componenti (dati locali e globali di sistema)
- - flusso dei dati attraverso i componenti (data dominant)

- flusso del controllo attraverso i componenti (control dominant)

SA/SADT - SOFTECH
RSL/SAS - BMDATC

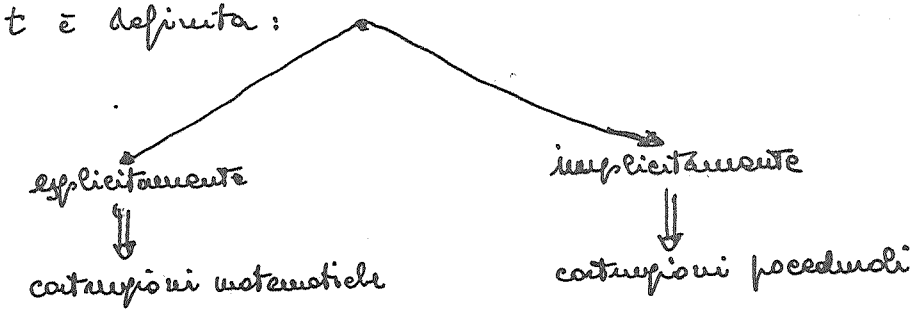
- Specifica funzionale

- Ad ogni componente si associa una funzione:

$t : X \to y$, $X \subset \Delta^* \times C^*$
$y \subset \Delta^* \times C^*$

- Δ = dominio dati
- C = dominio regoli di controllo (o stati)

- t è definita:



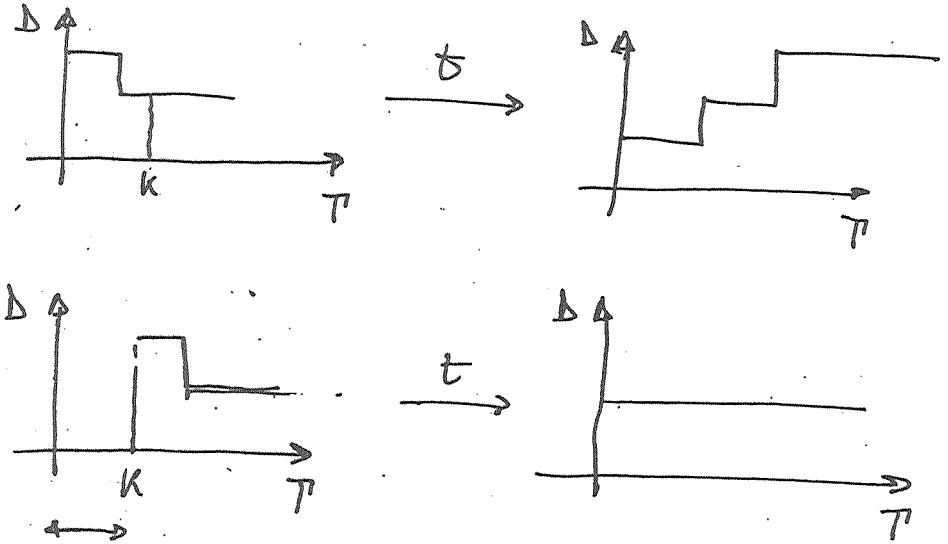
SPECIAL, PROLOG, OBJ, LISP, SETL, ---
(LISP, SETL: implicita)

- In ritemi real-time la funzione t risulta:

$$t : X \rightarrow Y$$

$$X \subset D_T \times C_T \quad , \quad Y \subset D_T \times C_T$$

- $D_T = T \rightarrow D^*$
- $C_T = T \rightarrow C^*$



SPECIAL, PROLOG, OBJ (esplicita) LISP (implicita)

- Progetto - Manutenzione

Vari modi di essere orientati

- linguaggio usato in un ambiente di strumenti di supporto ad una metodologia di sviluppo completa
  - RSL/SDS - BMDATC
  - SA/SADT - SOFTECH
- linguaggio ad ampio spettro
  - SPECIAL/HDM - SRI
  - SAFE
  - CIP

Tecniche di specifica

- Modello con cui il sistema è descritto.

- Più diffuse:

  - Primoli (RLP - GTE Lab.)
    (sistema è un insieme di behaviours)

  - Input - Output (LISP, PROLOG, OBJ)
    (sistema è un insieme di funzioni)

  - Stato (SPECIAL - SRI, NPN - Honeywell)
    (insieme di funzioni con stato)

Metodo di specifica

- Struttura con cui la specifica è espressa

- Più diffusi:

  - grafico (SA/SADT - SOFTECH, RSL/SAS - BMDATC)

  - assiomatico (PROLOG, OBJ) } AFFIRM

  - operazionale (LISP, SETL) } AFFIRM

Vantaggi del metodo proprio

- specifica architetturale di sistemi con caratteristiche (del controllo) molto diverse.
- profili sono strutture astratte, quindi una definizione espressa in profili è astratta.
- esiste una grande quantità di tecniche per l'analisi di profili.
- i profili offrono un supporto visivo alla comprensione della loro struttura.

Vantaggi del metodo automatico

- specifica funzionale
- metodi formali di prova
- eseguibilità
  ⇓
- prototipazione
- gli automi offrono un supporto mentale alla comprensione e definizione del sistema.

Vantaggi del metodo operazionale

- "WICKED SYSTEMS" di Rittel
  - Ogni formalizzazione corrisponde alla formulazione di una soluzione-implementazione
  - convergenza dipende dalla capacità a dare un'implementazione
  - Non esiste una soluzione definitiva
  - Non esiste una soluzione corretta
- Tutti i sistemi sono un po' WICKED ?!

Linguaggi di specifica

Linguaggi ad AMPIO SPETTRO

- L.K insieme di costrutti interpretivi dei costrutti definenti L.K-1.

  SPECIAL / HOM
  AFFIRM

- Unico nucleo di costrutti per le diverse fasi
  +
  strumenti di trasformazione di espressioni

  SAFE
  CIP
  SISTEMI PER LA SINTESI di PROGRAMMI

linguaggi	sistemi sviluppati	fasi descrivibili	tecniche	metodi
SA - SADT (SOFTECH INC)	Data processing concurrent (REAL-TIME) office	• Requirements • Architettura	Input/output	grafico
RSL - SDS (BMDATC)	Real-time	• Requirements • Architettura • funzionalità • progetto - man.	Input/output	Naturale strutturato + grafico
PSL - PSA (ISDOS)	Data processing	• Requirement • Architettura • funzionalità • progetto - man.	Input/output	Naturale strutturato + grafico
RLP (GTE lab.)	Real-time	funzionalità	stimoli + stato	assiomatico + grafico
OBJ	Data processing	• funzionalità • progetto - man	Input/output	assiomatico (algebrico)
PROLOG	Data processing	• funzionalità	Input/output	assiomatico (logico)
SPECIAL (HDM)	Data processing	• architettura • funzionalità • progetto - man • implementaz. AMPIO SPETTRO	Input/output + stato	assiomatico + [meccanismi] di struttura del controllo

AMBIENTI DI SVILUPPO SOFTWARE

SOMMARIO:

- OBIETTIVI
- LA VECCHIA CONCEZIONE
- GLI AMBIENTI INTEGRATI
- GLI STRUMENTI
- I META-AMBIENTI
- L'INTERFACCIA UOMO-MACCHINA

Dott. ROBERTO BARBUTI

AMBIENTI DESTINATARI:

quelli in cui l'attività principale è produrre software.

PROCESSO DI PRODUZIONE:

PROGETTO (SPECIFICA ...)
IMPLEMENTAZIONE
VERIFICA
DOCUMENTAZIONE
MANUTENZIONE

- INDUSTRIA PRODUTTRICE
- SOFTWARE HOUSE
- UNIVERSITA'

PROGETTI SOFTWARE

- PRODOTTI INDUSTRIALI
  caratterizzati da - AFFIDABILITA'
  - COSTI IL PIÙ POSSIBILE CONTENUTI
- PROGETTI SPERIMENTALI (DI RICERCA)
- PROGETTI SVILUPPATI COME ATTIVITA' DI FORMAZIONE
- PROGETTI SVILUPPATI DA UTENTI

OBIETTIVO DEGLI AMBIENTI DI SVILUPPO

MIGLIORARE LA PRODUZIONE.

- COSTI DELLO SVILUPPO
- QUALITA' DEL PRODOTTO

ATTRAVERSO

- FLESSIBILITA' DELL'ORGANIZZAZIONE DEL LAVORO
- MIGLIORAMENTO DEGLI STRUMENTI DI PRODUZIONE

- LINGUAGGIO DI PROGRAMMAZIONE AD ALTO LIVELLO
- AMBIENTE INTERATTIVO CON STRUMENTI INTEGRATI
- AMBIENTE CHE GESTISCA LA COOPERAZIONE

IMPORTANZA DEL LINGUAGGIO AD ALTO LIVELLO

- FACILITA' D'USO (orientato ai problemi)
- EFFICIENZA (se possibile adeguati strumenti, es: compilatore ottimizzante)
- AFFIDABILITA' (costrutti orientati alla programmazione "strutturata")
- DOCUMENTABILITA'
- FACILITA' DI MANUTENZIONE
- PORTABILITA'

AMBIENTE DI SVILUPPO SOFTWARE

AMBIENTE BATCH

- TANTE FASI SCOLLEGATE
- NESSUNO STRUMENTO DI DEBUGGING
- NESSUNA POSSIBILITA' DI INTERAZIONE DIRETTA CON GLI STRUMENTI
- NUMEROSI E COSTOSI CICLI PER LA MESSA A PUNTO

- MAGGIORE INTEGRAZIONE TRA LE DIVERSE FASI (attraverso il calcolatore)
- LE FASI SONO ANCORA VISTE COME LOGICAMENTE SEPARATE
- POSSONO ESISTERE STRUMENTI DI DEBUGGING

AMBIENTI INTEGRATI

- INTEGRAZIONE
  INTERFACCIA COMUNE A TUTTI GLI STRUMENTI
  MUTUA INVOCABILITA'
- INTERATTIVITA'
  INCREMENTALITA' DEGLI STRUMENTI
- GRANULARITA'
- ESTENDIBILITA'
  NUOVI STRUMENTI
  ADATTAMENTO AI DIVERSI AMBIENTI

INTERFACCIA TRA GLI STRUMENTI (FORMA INTERNA DEI PROGRAMMI)

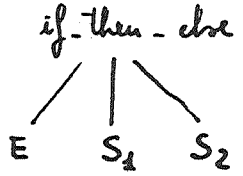
IN GENERE

RAPPRESENTAZIONE DELLA SINTASSI ASTRATTA + ATTRIBUTI

- ELIMINAZIONE DI TUTTE LE PARTI NON SIGNIFICATIVE SEMANTICAMENTE (delimitatori, categorie sintattiche ...)
- EVIDENZA DEI COSTRUTTI E DEI LORO ARGOMENTI

Es:

if E then S_1 else S_2



GLI STRUMENTI

- STRUMENTI DI SCRITTURA, MODIFICA E VISUALIZZAZIONE DI PROGRAMMI (EDITOR)

- STRUMENTI DI ANALISI STATICA
  VERIFICATORE REGOLE DI SCOPE
  VERIFICATORE CORRETTEZZA DEI TIPI
  ANALIZZATORE USO DELLE VARIABILI

- STRUMENTI PER L'ANALISI DINAMICA
  INTERPRETE STANDARD
  INTERPRETE SENZA CONTROLLO DEI TIPI
  DEBUGGER

- STRUMENTI DI TRASFORMAZIONE
  COMPILATORE (anche solo generatore di codice)
  OTTIMIZZATORE

- STRUMENTI DI GESTIONE DI PROGETTI
  BASE DI DATI DI PROGETTO

EDITOR

FUNZIONALITA':

- CREAZIONE DI PROGRAMMI
- COSTRUZIONE DELLA FORMA INTERNA
- VISUALIZZAZIONE (PRETTY PRINTING)
- MODIFICA

GUIDATO DALLA SINTASSI

(conosce la sintassi del linguaggio e in grado di rilevare errori durante la stesura)

DUE MODI DI OPERARE:

ANALITICO

- CONTIENE UN ANALIZZATORE SINTATTICO INCREMENTALE.
- PERMETTE DI SCOPRIRE ERRORI SINTATTICI ALL'ATTO DELLA STESURA.

GENERATIVO

- GENERA AUTOMATICAMENTE LA STRUTTURA DEI COSTRUTTI
- NON HA BISOGNO DI UN ANALIZZATORE SINTATTICO

ES. DI SCRITTURA IN MODO GENERATIVO

IF

IF <EXPR> THEN <STAT> ELSE <STAT>

AND

IF <EXPR> AND <EXPR> THEN <STAT> ELSE <STAT>

INCREMENTALITA'

NELL'ALBERO DI SINTASSI ASTRATTA GENERATO POSSONO RIMANERE CATEGORIE SINTATTICHE.

MODO ANALITICO:
PRESENTA SVANTAGGI PER CHI NON CONOSCE ESATTAMENTE LA SINTASSI DEL LINGUAGGIO

MODO GENERATIVO
PUO' ESSERE MOLTO LENTO E DIFFICILE DA USARE (IN PARTICOLARE SU ESPRESSIONI)

E' CONVENIENTE AVERE UN EDITOR CHE POSSA FUNZIONARE IN ENTRAMBI I MODI.

STRUMENTI DI ANALISI STATICA

INTERPRETI DEI PROGRAMMI SU DOMINI NON-STANDARD.

SONO LE COMPONENTI DELL'ANALISI SEMANTICA DEI COMPILATORI.

POSSONO ESSERE APPLICATI INDIPENDENTEMENTE.

- SCOPER
- TYPE-CHECKER
- SET-USE ANALYSER
- INTERFACE CHECKER

⋮

INTEGRAZIONE:
E' UTILE POTER CHIAMARE L'EDITOR DA UNO DI QUESTI STRUMENTI (PROBLEMI PER QUANTO RIGUARDA LA CONSISTENZA DELL'ANALISI)

INCREMENTALITA':
POSSONO ESSERE USATI SU PROGRAMMI NON COMPLETAMENTE SPECIFICATI. (AGGIUNGONO INFORMAZIONI AI PROGRAMMI PER ANALISI SUCCESSIVE)

STRUMENTI DI TRASFORMAZIONE

GENERATORE DI CODICE

- FUNZIONAMENTO DEI SISTEMI SU MACCHINE DIVERSE
- PROBLEMI HOST - TARGET

OTTIMIZZATORE

- SISTEMI EFFICIENTI

LA BASE DI DATI DI PROGETTO

ORGANIZZA E GESTISCE OGNI INFORMAZIONE RELATIVA AI PROGETTI SVILUPPATI

- CONTIENE LE VARIE VERSIONI DEI MODULI COMPONENTI IL PROGETTO
  - VERSIONI VERTICALI (TEMPORALI)
  - VERSIONI ORIZZONTALI (DIVERSE IMPLEMENTAZIONI DELLO STESSO MODULO)
- MANTIENE LA CONSISTENZA TRA I MODULI
- AIUTA L'UTENTE NELLA DOCUMENTAZIONE
- TIENE TRACCIA DELLE ANALISI E DELL'USO DEI MODULI
- AIUTA L'UTENTE NELLA GESTIONE AMMINISTRATIVA DEL PROGETTO
  - PREPARAZIONE DI DOCUMENTI
  - PIANIFICAZIONE
  - ⋮

IL LINGUAGGIO DI COMANDI

- IL PIÙ POSSIBILE SEMPLICE E FACILE DA USARE
  - BUONA INTERFACCIA

&

- POSSIBILITA' DI POTERLO USARE DA PROGRAMMA

SOLUZIONE POSSIBILE

LO STESSO LINGUAGGIO IN CUI SONO SCRITTI GLI STRUMENTI CON UN MAGGIORE ZUCCHERO SINTATTICO

ES:
- PARAMETRI DI DEFAULT
- MENU
- ⋮

STRUMENTI DI ANALISI DINAMICA

OPERANO DIRETTAMENTE SULLA FORMA INTERNA DEI PROGRAMMI (RAPPRESENTAZIONE DELLA SINTASSI ASTRATTA)

E' SEMPLICE CONTROLLARE DIRETTAMENTE L'ESECUZIONE DEI PROGRAMMI. E' POSSIBILE RICHIAMARE ALTRI STRUMENTI QUALI EDITOR E DEBUGGER.

- INTERPRETE STANDARD
- INTERPRETE SENZA CONTROLLI STATICI
- INTERPRETE SIMBOLICO

COME PER GLI ALTRI STRUMENTI E' CONVENIENTE CHE SIANO INCREMENTALI.

(GESTIONE COMPLESSA DELLO STATO DI UN PROGRAMMA).

NON E' INDISPENSABILE L'EFFICIENZA

IL DEBUGGER

CONSENTE DI CONOSCERE E ALTERARE LO STATO DI UN PROGRAMMA DURANTE LA SUA ESECUZIONE. (ATTRAVERSO UN QUALSIASI INTERPRETE, STATICO O DINAMICO)

CARATTERISTICHE:

- INSERZIONE E RIMOZIONE DI BREAKPOINTS
- TRACCIA DEL FLUSSO
- STATISTICHE
- FUNZIONAMENTO IN SINGLE-STEP
- ISPEZIONE DELLO STATO
- MODIFICA DELLO STATO ATTRAVERSO L'ESECUZIONE DIRETTA DI FRASI DEL LINGUAGGIO
- VALUTAZIONE DI ASSERZIONI
- CONSISTENTI MODIFICHE DEL PROGRAMMA ATTRAVERSO L'EDITOR

INDIPENDENZA DAGLI STRUMENTI OTTENUTA ATTRAVERSO LA DEFINIZIONE DEI DATI SUI QUALI OPERANO GLI INTERPRETI CON TIPI DI DATO ASTRATTI

META-AMBIENTI

OVVERO I GENERATORI DI AMBIENTI

AMBIENTI GENERALI FACILMENTE ADATTABILI AD UN PARTICOLARE LINGUAGGIO

COSTITUITI DA:

- GENERATORE DI EDITORS
- METODOLOGIA DI DERIVAZIONE DI STRUMENTI DI ANALISI DALLA DEFINIZIONE DEL LINGUAGGIO
- STRUMENTO GENERALE DI DEBUGGING
- BASE DI DATI FACILMENTE ADATTABILE

GENERATORE DI EDITOR

ARGOMENTI:

SINTASSI CONCRETA
SINTASSI ASTRATTA
MAPPING TRA LE DUE

RISULTATO:

EDITOR GUIDATO DALLA SINTASSI PER IL LINGUAGGIO

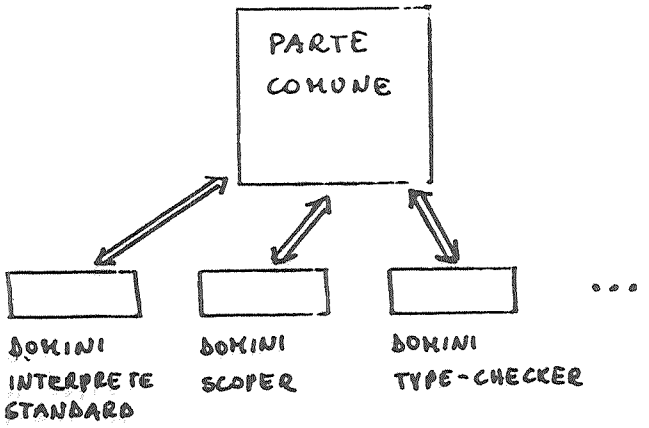
(ANALITICO E/O GENERATIVO)

METODOLOGIA DI DERIVAZIONE DI STRUMENTI DI ANALISI

DEFINIZIONE FORMALE DEL LINGUAGGIO
ES: IN STILE DENOTAZIONALE
(INTERPRETE STANDARD)

ALTRI STRUMENTI OTTENUTI CAMBIANDO LA DEFINIZIONE DI ALCUNI DOMINI
(ES: ELIMINAZIONE DEI VALORI PER IL CONTROLLO DEI TIPI)

TRASFORMAZIONI DAL LINGUAGGIO DI DEFINIZIONE AL LINGUAGGIO IN CUI E' REALIZZATO L'AMBIENTE.
(MEDIANTE MECCANISMI AUTOMATICI)



STRUMENTO GENERALE DI DEBUGGING

UTILIZZABILE PER LINGUAGGI CON GLI STESSI DOMINI SEMANTICI (ES AMBIENTE E MEMORIA)

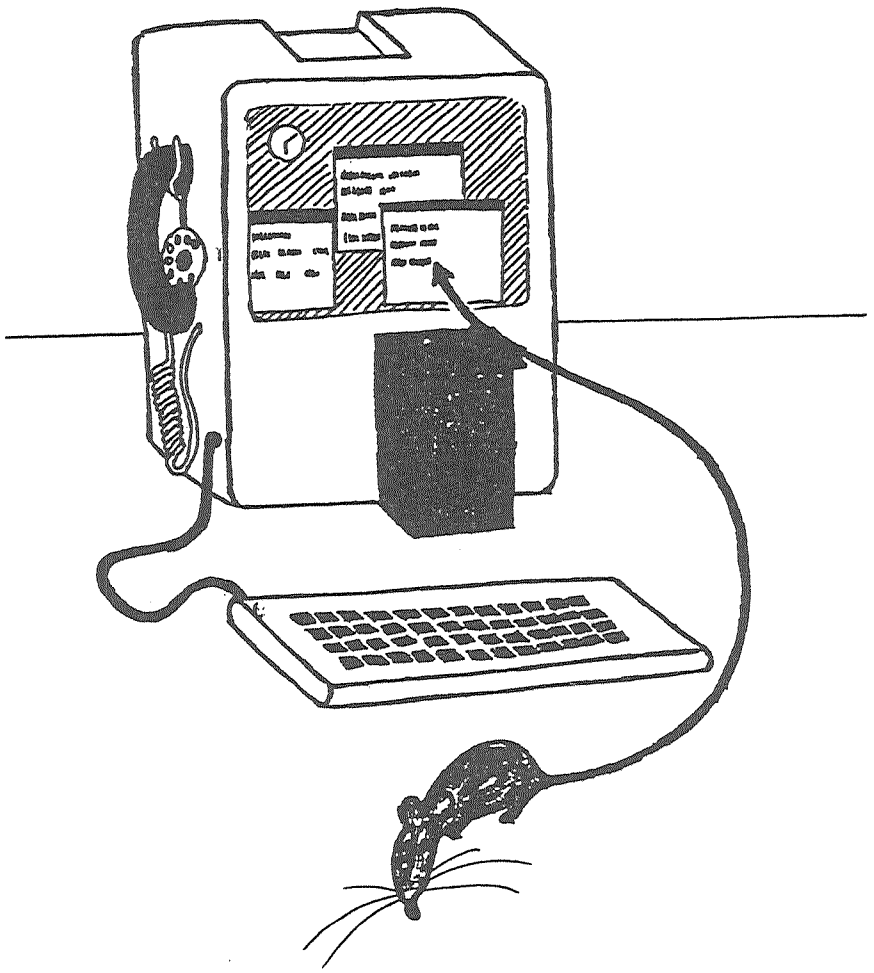
POCHE MODIFICHE PER QUANTO RIGUARDA L'ADATTAMENTO ALLA SINTASSI DEL LINGUAGGIO

BASE DI DATI DI PROGETTO

ADATTABILE ALLE SPECIFICHE INTERFACCE DEL LINGUAGGIO

INTERFACCIA UOMO - MACCHINA

- GESTIONE DI TUTTE LE ATTIVITA' DEL PROGETTISTA. (INTEGRAZIONE CON UN SISTEMA DI AUTOMAZIONE DI UFFICI)
- LINGUAGGIO DI COMANDI A MENU'
- POSSIBILITA' DI PIU' PROCESSI CONTEMPORANEI
- RETE DI COMUNICAZIONE PER GESTIRE MODULI SVILUPPATI SU MACCHINE DIVERSE



CNUCE - Istituto del Consiglio Nazionale Delle Ricerche, Pisa.
Corso "I Linguaggi di Specifica per la Produzione del Software".
Pisa, 5-7 novembre 1984.

Seminario dal titolo

Prospettive dei Metodi di Specifica Formale nella Produzione di Software

Diapositive per la retroproiezione.

Prof. Ugo Montanari
Dipartimento di Informatica
Università di Pisa.

Modelli di ciclo di vita del software

- una serie di approssimazioni successive
- versioni adatte a particolari applicazioni

1ª appr.: programmare = scrivere il programma

2ª appr: specifica
design
codifica
solo produzione

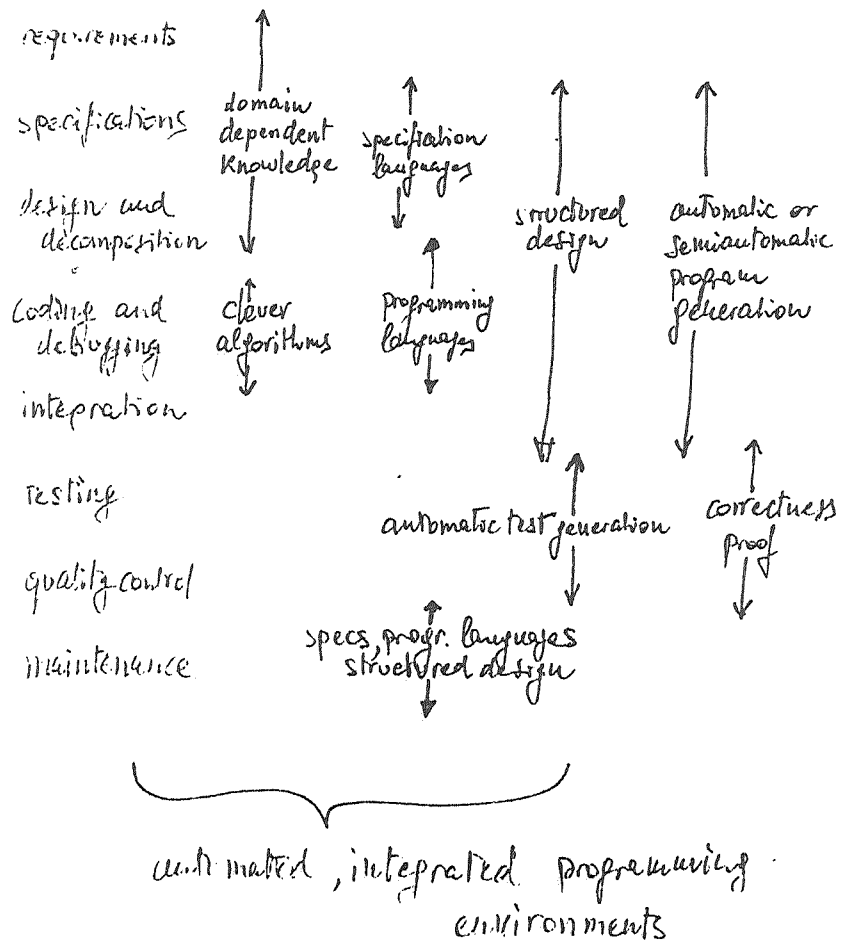
3ª appr: - produzione come fasi successive di passaggio dalla specifica alla implem.
- manutenzione
} lo stesso comportamento esterno

4ª appr.: requirements specifications
⋮
global testing
maintenance

- verifica di consistenza rispetto alle fasi precedenti
- validazione rispetto ai requirements

Il ciclo di vita del software

(2)

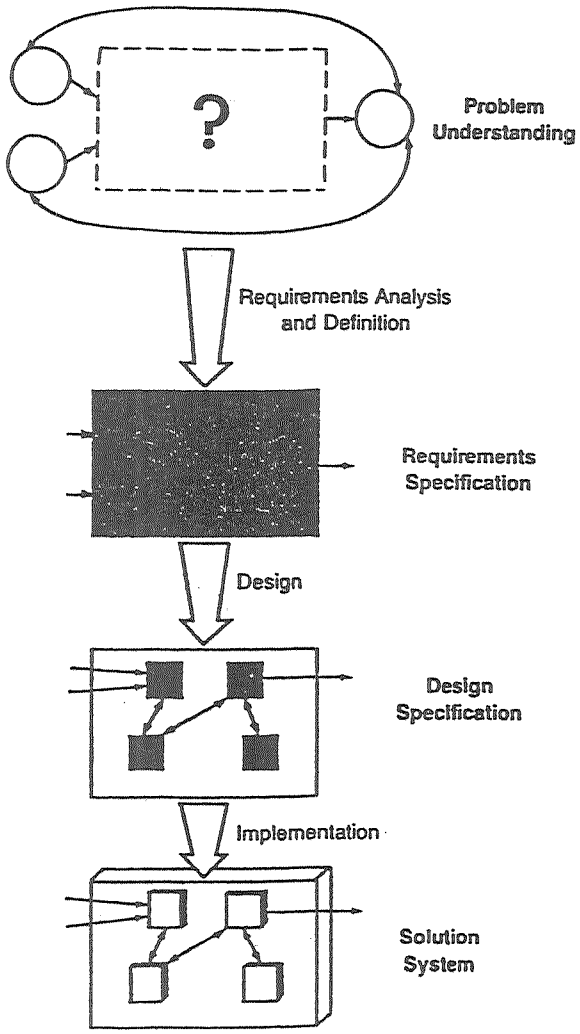


Limitazioni del ciclo di vita classico

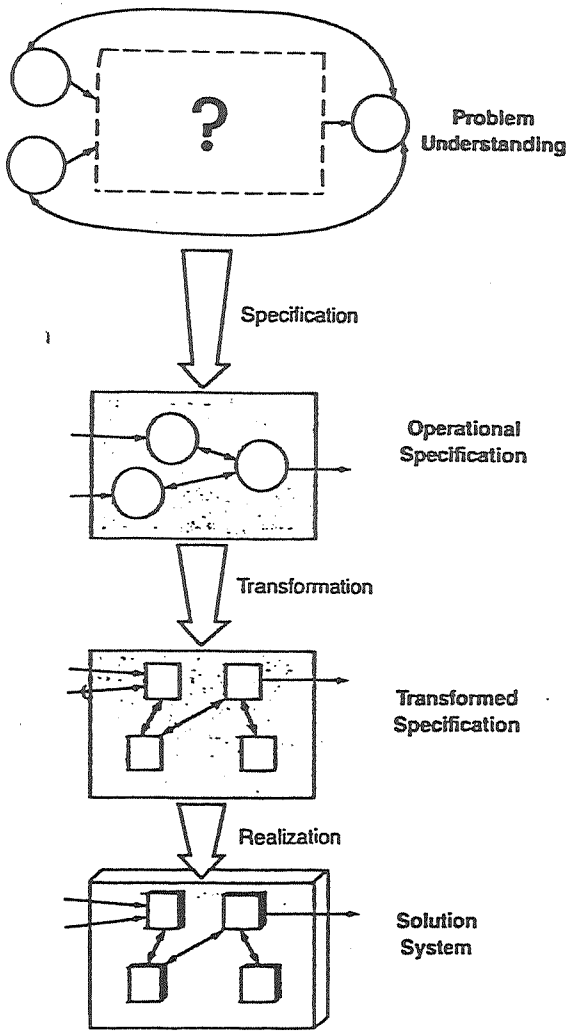
informale nelle prime fasi
- test in linguaggio naturale
- metodologie basate su "schede di caratteristiche"
- solo il linguaggio di programmazione è completamente formale

inadatto ad affrontare problemi totalmente nuovi
- decisioni funzionali prima delle strutturali
- decisioni globali fallimentari ma controllate (blocchi)
→ metodo operazionale (es. fast prototyping)

la documentazione per la manutenzione descrive solo il programma
- è necessario descrivere il processo di progettazione in modo da poter risalire al livello giusto per le modifiche necessarie
- distinzione tra il vero e un processo pulito senza errori e valutazioni di costo



The conventional approach to software development is based on the principle of top–down decomposition of black boxes, and all its features can be derived from that philosophy.



The operational approach to software development is based on separation of problem-oriented from implementation-oriented concerns, and all its features can be derived from that philosophy.

Advantages of the Conventional and Operational Approaches*

	Conventional	Operational
Validation	Informal requirements can be understood directly by everyone	Formal specifications are rigorous, can be analyzed formally User has a model of the system to internalize and evaluate Operational approach makes rapid prototyping available automatically
Verification		Both testing and program-proving have well-known inadequacies Transformational implementation guarantees correctness
Automation		Transformation and realization are highly automatable
Maintenance		Operational approach addresses directly the structural conflict between efficiency and ease of maintenance
Management	Conventional approach provides organizationally useful milestones Managerial aspects of operational approach unexplored	Executable specification provides early results and accountability

* Assuming that both approaches could be applied successfully to any project, the differing properties of the two approaches would still cause differences in the handling of certain chronic problems. Here advantages of the two approaches are summarized.

Validation: building the Right System
Verification: building the System Right

(6)

Weaknesses of the Conventional and Operational Approaches

Conventional	Operational
Conventional approach does not allow for the effect of system structure on system behavior	Operational approach presents a danger of premature design decisions
"Black-box" requirements are very difficult to specify	It may prove difficult to execute specifications with adequate performance
Top–down decomposition is difficult and risky	Transformational implementation is a new and undeveloped technology People may find it hard to guide transformations

* It may not be possible to apply both approaches successfully to any project. Here are the known feasibility problems on each side.

(7)

Le principale differenze tra descrizione e operazione è
il modo in cui le decisioni vengono ordinate

Conversazione: decisioni guidate prima dalla struttura
può essere una limitazione

Se metodo descrive e descrivere in modo
dichiarativo (e quindi può essere diverso dal
contesto usa metodologie per il primo principio

Il metodo operazionale presenta visioni

Esempio

Scelta decisioni implicite

- decisioni di decomposizione non sempre esplicite
- decisioni più globali, ma è complesso da specificare

Scelta decisioni: i.e. più semplice decisioni: le loro sottili errori-prone

- decisioni di cambio top dominano tutte le proprietà del sistema

Some Instances of the Operational Approach

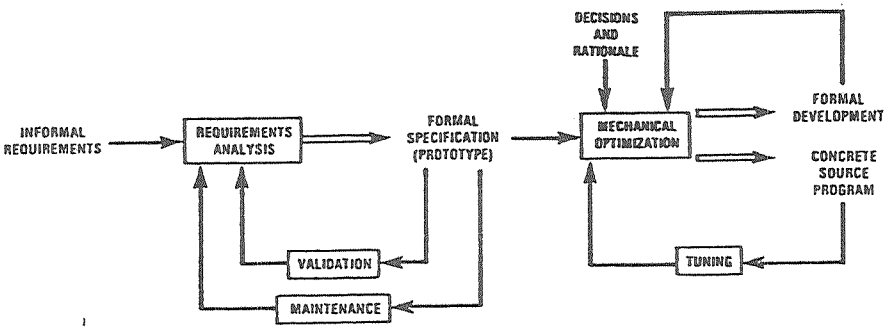
	Jackson System Development Method	PAISLey Project	Gist Project	Applicative Programming
Domain of Origin	Data processing	Real-time systems	AI and database systems	Programming
Special Features	Specific method for obtaining operational specification	Explicit representation and transformation of concurrency and distribution	Very high-level specification language motivated by expressive power of English	Strong mathematical foundations
	Maintainability stressed	Formal and executable performance constraints	Associative retrieval and historical reference	Historical emphasis on efficient execution of the operational specification itself, including special-purpose hardware architectures
	Operational specification uses implementation language, transformations operate within implementation language	Execution of incomplete specifications	Use of constraints to refine nondeterminism (as in common-sense reasoning)	
Automated Support	Specifications informal, not executable	Complete integrated environment possible and being planned	Complete integrated environment possible and being planned	Specification environments exist
	Transformations language-dependent			Transformations being automated
Current Status	In real-world use	First tools being implemented	Many tools implemented (including natural-language paraphraser, behavior explainer)	Transformations and special-purpose hardware in research stage, use without them stable
	Methods in research stage			

Miglioramenti previsti a breve termine

- derivazione manuale del programma, ma strettamente guidata dalle specifiche
  - esempio: compilatore Ada della DOC
- integrazione (limitata) tra le varie fasi
  - ad esempio: - unica rappresentazione interna
    - editors guidati dalla sintassi (delle sintassi)
    - unico linguaggio di configurazione
  - allocazione degli strumenti su un'unica workstation
    - Es. Grespin
  - strumenti basati su "sistemi esperti"
    - Es. approccio innovativo in STARS.

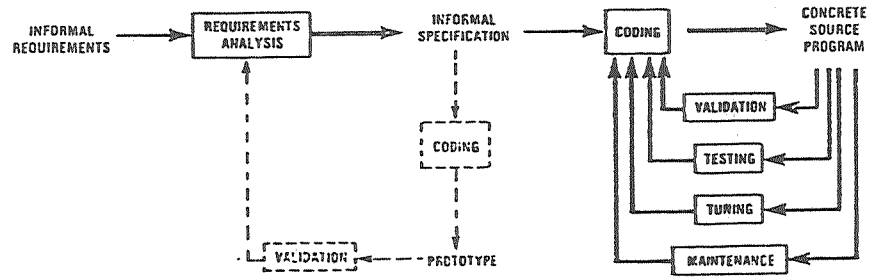
Estimated DoD software initiative impact on software productivity.

Cost Driver	DoD Average-Effort Multipliers 1976	1983	1993
Use of Software Tools	1.05	1.02	0.85
Use of Modern Programming Practices	0.98	0.95	0.85
Programming Language Experience	1.03	1.02	0.98
Software Environment Experience	1.05	1.03	0.95
Computer Execution Time Constraint	1.25	1.18	1.11
Computer Storage Constraint	1.22	1.15	1.06
Computer Turnaround Time	1.03	1.01	0.90
Reduced Requirements Volatility	1.17	1.15	1.00
Retool Avoidance	1.06	1.06	1.00
Software Reuse	0.93	0.90	0.50
Relative Effort	2.01	1.54	0.36
Productivity Gain		1.30	4.34



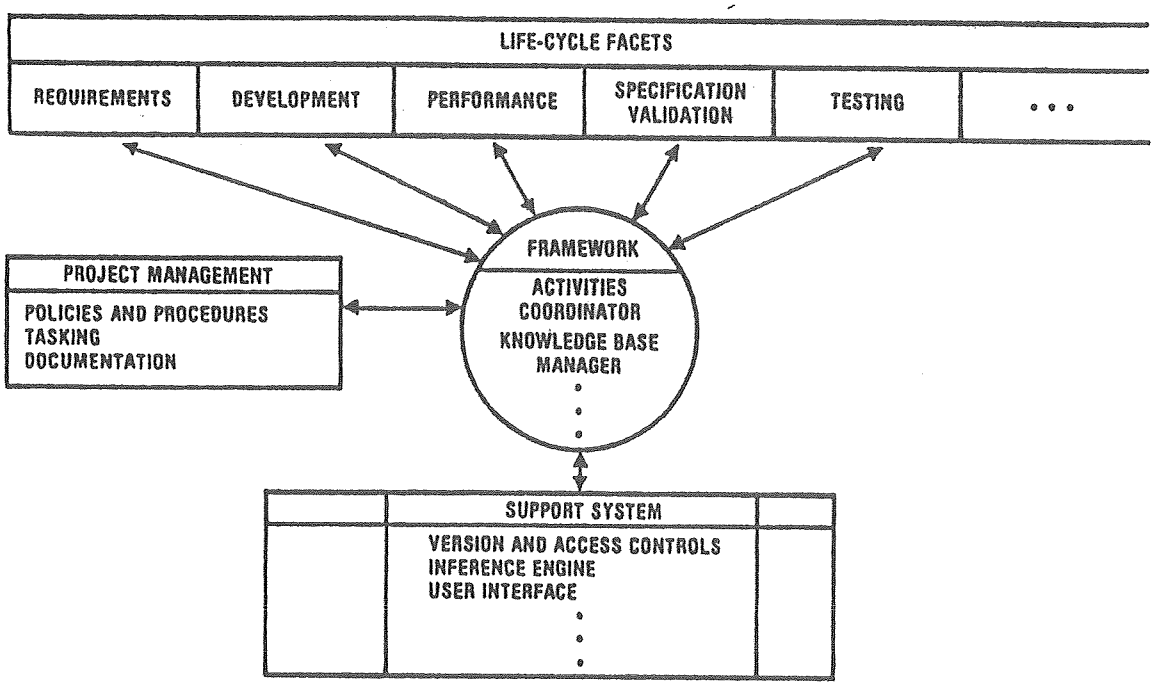
(a)

AUTOMATION-BASED PARADIGM	CURRENT PARADIGM
FORMAL SPECIFICATION	INFORMAL SPECIFICATION
PROTOTYPING STANDARD	PROTOTYPING UNCOMMON
SPECIFICATION IS THE PROTOTYPE	PROTOTYPE CREATED MANUALLY
PROTOTYPE VALIDATED AGAINST INTENT	CODE VALIDATED AGAINST INTENT
PROTOTYPE BECOMES IMPLEMENTATION	PROTOTYPE DISCARDED
IMPLEMENTATION MACHINE AIDED	IMPLEMENTATION MANUAL
TESTING ELIMINATED	CODE TESTED
FORMAL SPECIFICATION MAINTAINED	CONCRETE SOURCE CODE MAINTAINED
DEVELOPMENT AUTOMATICALLY DOCUMENTED	DESIGN DECISIONS LOST
MAINTENANCE BY REPLAY	MAINTENANCE BY PATCHING



(b)

Paradigm comparison: (a) automation-based paradigm and (b) current paradigm.



Generalized knowledge-based software assistant structure. (Adapted from Green et al., "Report on a Knowledge-Based Software Assistant," Technical Report KES. U.83.2, Kestrel Institute, Palo Alto, Calif., July 1983.)

(13)

Miglioramenti a medio - lungo termine basati sull'uso di metodi formali

- Definizione formale della semantica di tutti i linguaggi utilizzati nel ciclo di vita del software (es. attributi, relazioni, ecc.)
- Mix di linguaggi di specifica adatti alle varie esigenze (aspetto logico)
- Eseguibilità dei linguaggi di specifica per il rapid prototyping
- Descrizione formale del processo di trasformazione delle informazioni software (interconnessione profilazione)
- Linguaggi per l'intero ciclo di vita (wide spectrum languages)
- Strumenti di supporto

(14)

Pisa 6.11.1984

L'OBIETTIVO METOD DEL PROGETTO FINALIZZATO "INFORMATICA" C.N.R.

LINGUAGGI INFORMALI DI SPECIFICA

M.Maiocchi
Univ. Milano
Etnoteam spA

PROPOSTA GLOBALE DALLA STRUTTURA DEL PRODOTTO DELL'OBIETTIVO METOD

1. CICLO DI SVILUPPO DEL SOFTWARE
2. FASI DI SVILUPPO
   - 2.1 DEFINIZIONE DI REQUISITI
   - 2.2 DEFINIZIONE DELLE SPECIFICHE FUNZIONALI
   - 2.3 DEFINIZIONE ARCHITETTURALE DELLA STRUTT. DEI MODULI
   - 2.4 PROGRAMMAZIONE
   - 2.5 INTEGRAZIONE
3. ATTIVITÀ PARALLELE O TRASVERSALI ALLO SVILUPPO
   - 3.1 COSTRUZIONE DEGLI STRUMENTI DI TESTING
   - 3.2 CONTROLLO QUALITÀ
   - 3.3 DOCUMENTAZIONE
   - 3.4 MISURE E CONTROLLO AVANZAMENTI
   - 3.5 MANUTENZIONE
4. SPECIFICI AMBITI DI TIPO DI PRODOTTO SOFTWARE
   - 4.1 EDP
   - 4.2 SOFTWARE TELEFONICO
   - 4.3 SOFTWARE PER TEMPO REALE
   - 4.4 SISTEMI OPERATIVI
   - 4.5 SOFTWARE DIAGNOSTICO
5. ASPETTI DI FORMAZIONE
6. MECCANIZZAZIONE DI UNA FABBRICA DEL SW
7. ASPETTI DI SINTESI E ASPETTI AVANZATI
8. GLOSSARIO

ESIGENZE

STANDARDS DI GESTIONE

STANDARDS OPERATIVI

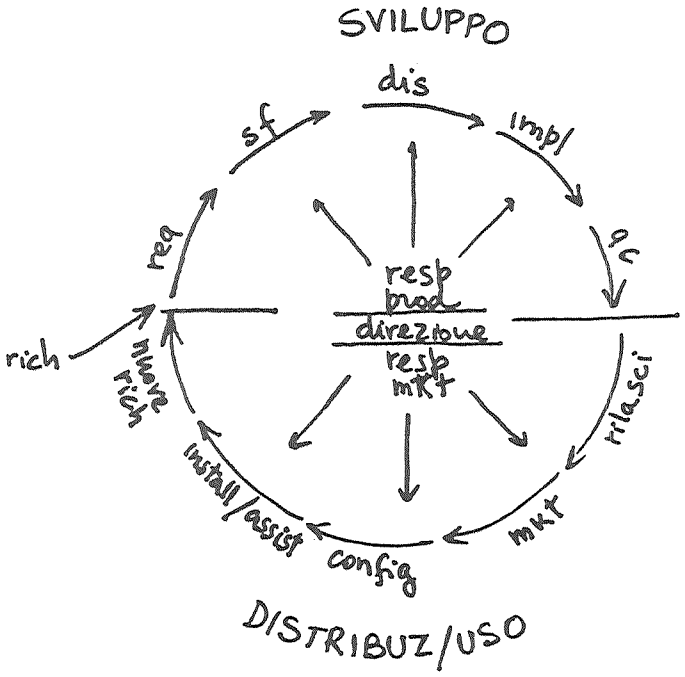
STANDARDS DI DOCUMENTAZIONE

STRUMENTI DI PIANIFICAZIONE

METODI

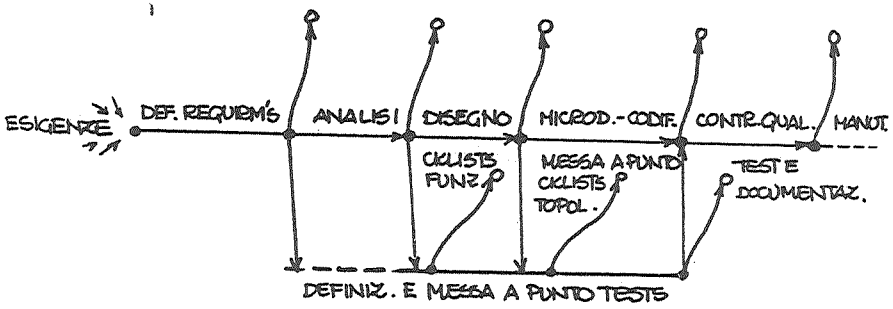
STRUMENTI

IL CICLO DI VITA

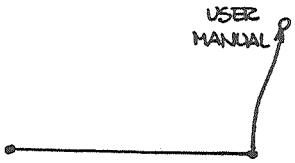


METOD

Sviluppo



Documenti User



Revisioni

Piani

DOCUMENTAZIONE

[illegible]

SR

SF

SD

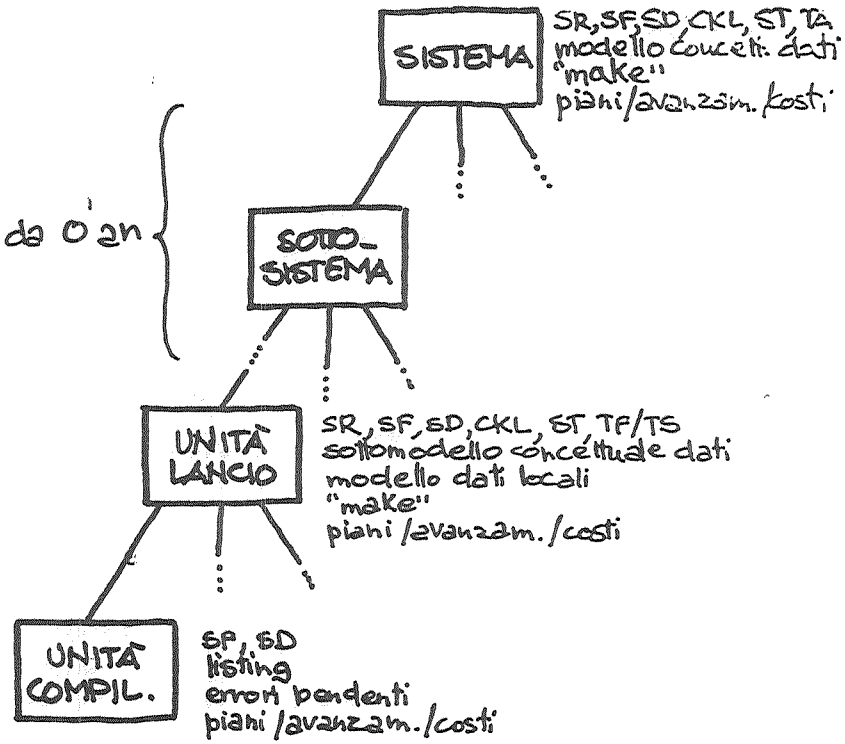
SC

CKL

ST

QCR

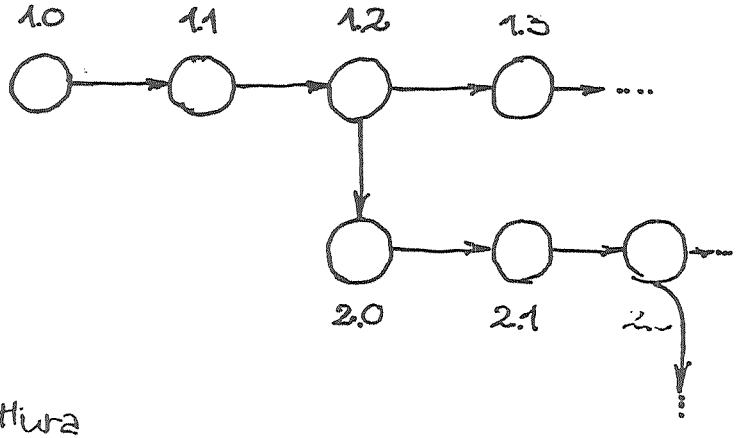
LA STRUTTURA DI UN PRODOTTO IN GESTIONE



SR specifiche requisiti
SF " funzionali
SD " disegno
CKL checklist
ST specifiche dei tests
TA tests di accettazione
TD " di sistema
TF " funzionali

STRUTTURA DI UN PRODOTTO E DELLA SUA EVOLUZIONE

Versione



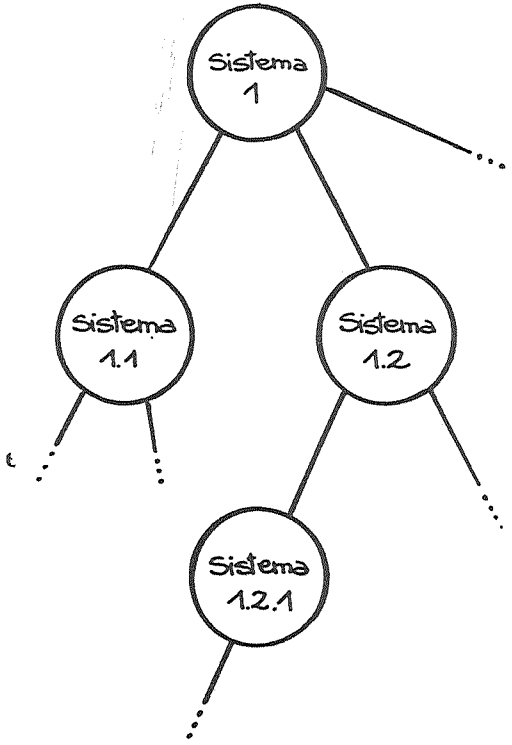
Struttura

Versione n. m : TOC

compon α : livello
compon β : "
" : "
. . . .

GESTIONE DI SISTEMI IN EVOLUZIONE

SCCS - like



SOFTWARE FACTORY

TOOLS PLURIFASE

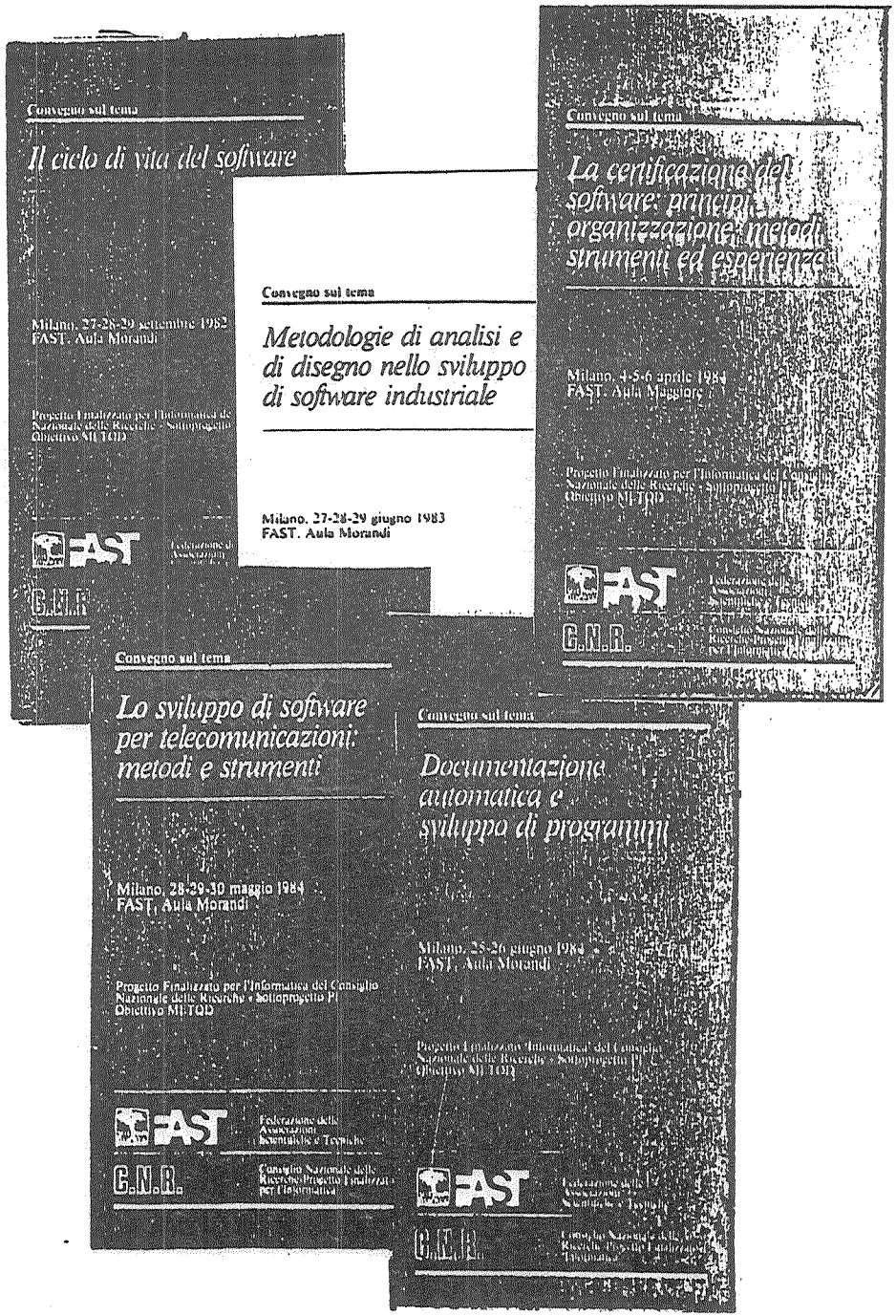
TOOLS MONOFASE

TEXT PROCESSING

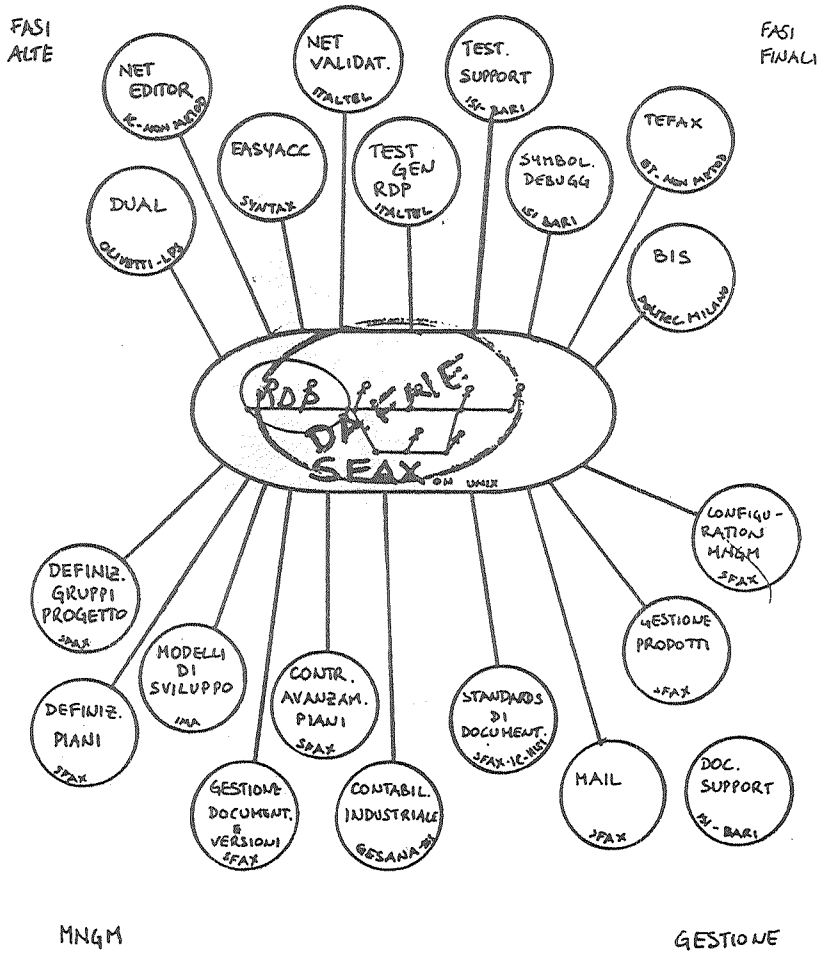
UNIX

EP13

I CONVEGNI



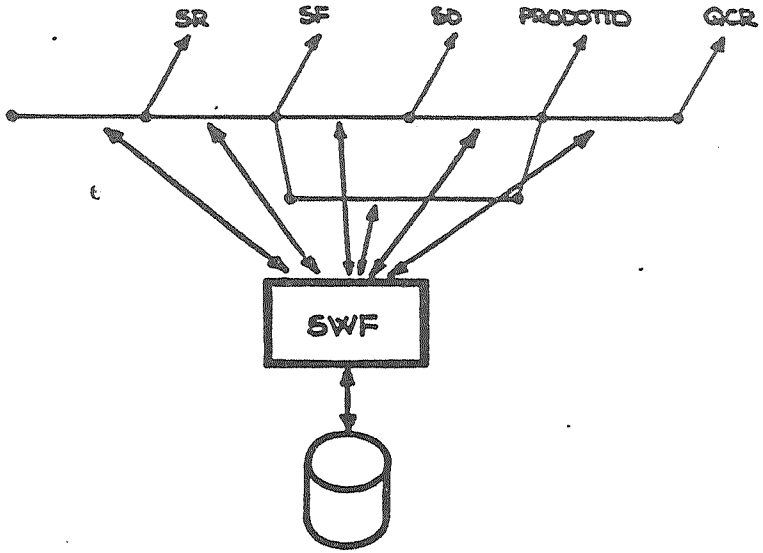
VERSO UNA SOFTWARE FACTORY



UNA SOFTWARE FACTORY

NON

MA



PIANIFICAZIONE

- Raccolta dati
- Modelli
- Pianificazione di dettaglio
- Controllo avanzamento
- Statistiche

REQUISITI E SPECIFICHE FUNZIONALI

- Documentazione
- Linguaggi formali (?)

DISEGNO E IMPLEMENTAZIONE

- Documentazione
- Macrogeneratori
- Compilatori
- Linker

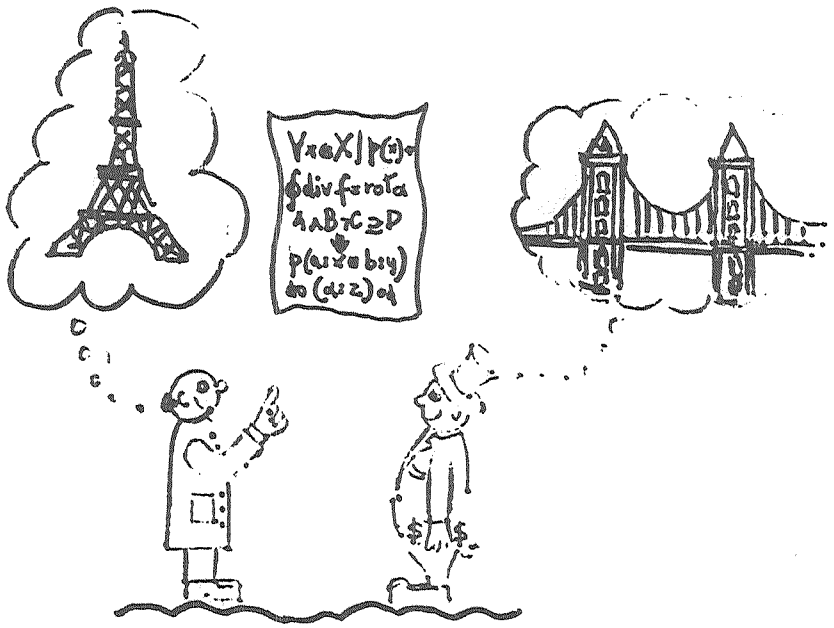
CONTROLLO DI QUALITÀ

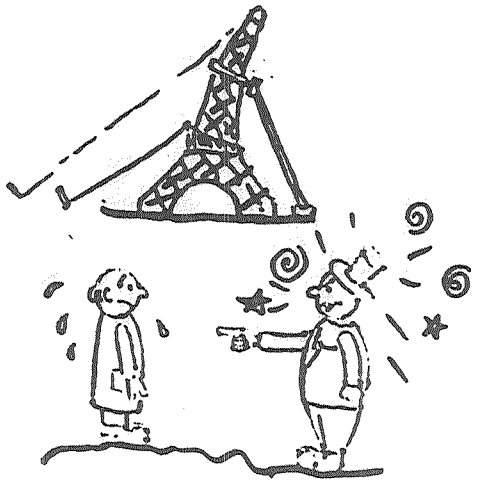
- Checklist
- Documentazione tests
- Preparazione tests
- Raccolta e analisi risultati

REQUIREMENTS FOR A METHOD FOR REQUIREMENTS ORIENTED TOWARDS THE COMMUNICATION WITH WHO REQUIRES

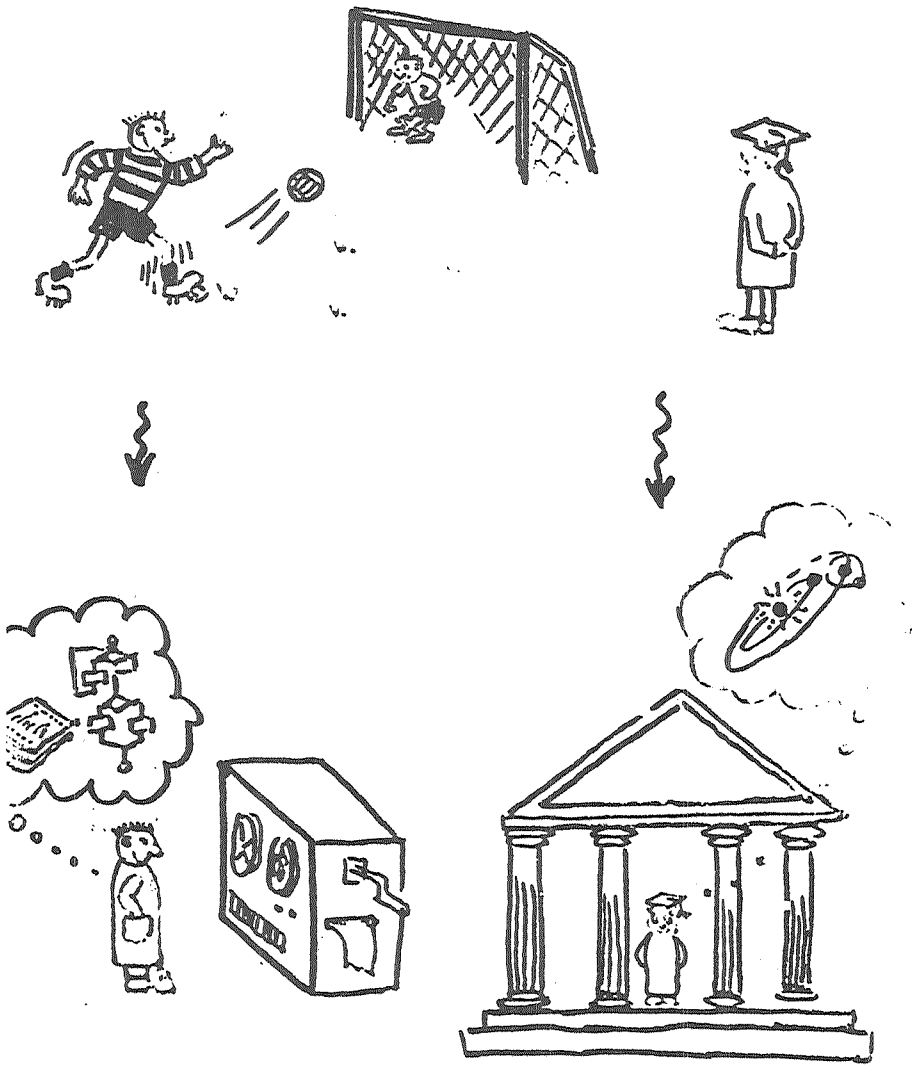
- UNDERSTANDABLE
- UNAMBIGUOUS, FORMAL
- USEFUL

UNDERSTANDABLE : TO .....





FORMAL



GΦ.4
LZ73

USEFUL

```
read n
i:=0
fact:=1
while n>i
    i:=i+1
    fact:=fact*i
endwhile
write fact
```

computes the result (you can't stop before the end)

```
read x
sqrt:=z
until |y-sqrt| < ε   you are tired
    y:=x/sqrt   (approx.)
    sqrt:=(y+sqrt)/2   (approx.)
enduntil
write sqrt
```

approximates the goal (you can stop, loosing some digits)

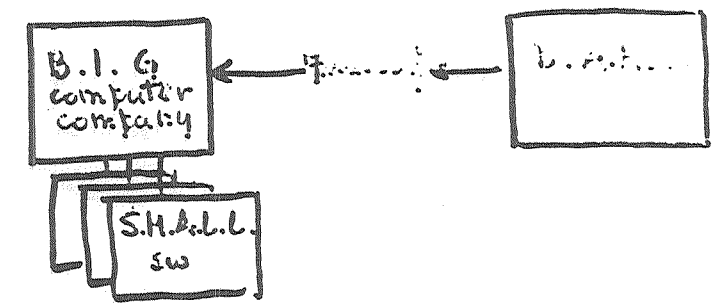
GΦ.5 / GΦ.5a
LZ71 LZ75

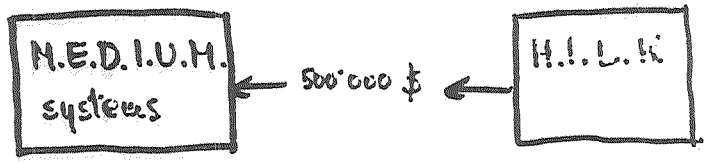
REQUIREMENTS FOR A METHOD FOR REQUIREMENTS ORIENTED TOWARDS THE COMMUNICATION WITH WHO REQUIRES

- UNDERSTANDABLE BY THE "COW BOY"
- UNAMBIGUOUS, FORMAL, BUT NOT "ASTONISHING" FORMALIZED
- PARTIALLY USEFUL WHEN PARTIALLY USED

GTØ8/ GTØ28
LZ 71 12 75

WHICH COMMUNICATION





GT Ø9

COST / BENEFITS

INFORMATION SYSTEMS ARE NOT A GOAL, BUT A WAY FOR INCREASING THE OPERATIONAL EFFICIENCY

↓

THE BENEFITS OF AN INFORMATION SYSTEM MUST BE MEASURED IN TERMS OF OPERATIONAL ADVANTAGES, EXPRESSED AS ECONOMICAL RESULT

ESIGENZE

AVERE A DISPOSIZIONE UN "LINGUAGGIO" PER DESCRIVERE E COSTRUIRE MODELLI DI "PROCEDURE"

- PARALLELISMO DI ATTIVITÀ
- ASINCRONIA TRA ATTIVITÀ
- DIPENDENZA O INDIPENDENZA
- CONSUMO, OCCUPAZIONE, PRODUZIONE DI RISORSE
- ATTIVITÀ UMANE O AUTOMATICHE

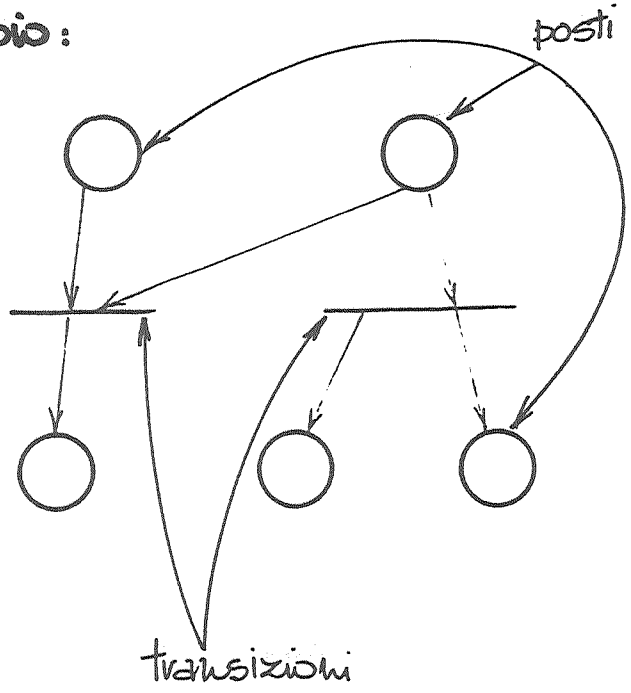
- LINGUAGGIO NON AMBIGUO
- SVILUPPO TOP-DOWN POSSIBILE

RETI DI PETRI

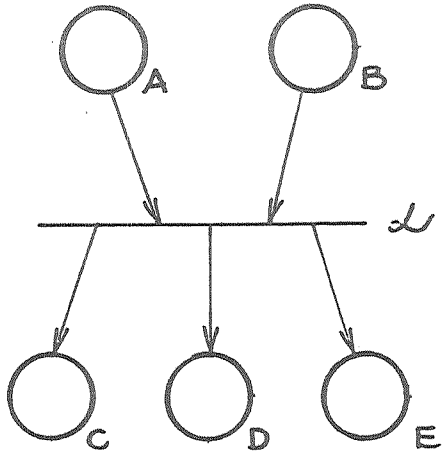
[illegible]
IN CUI [illegible]

POSTI
TRANSIZIONI

Esempio:

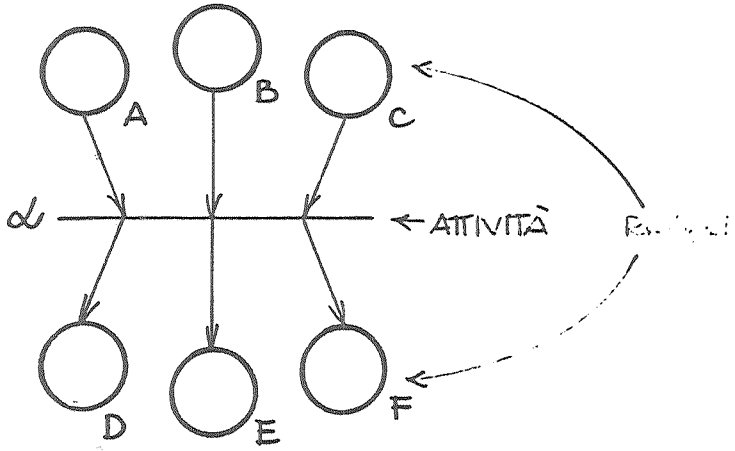


RETI DI PETRI
INTERPRETAZIONE



SE UN SISTEMA SI TROVA IN UNO STATO CARATTERIZZATO DA A E B, ESSO PUÒ EVOLVERE ATTRAVERSO LA TRANSIZIONE α PASSANDO IN UNO STATO CARATTERIZZATO DA C, D ED E.

RETI DI PETRI
INTERPRETAZIONE



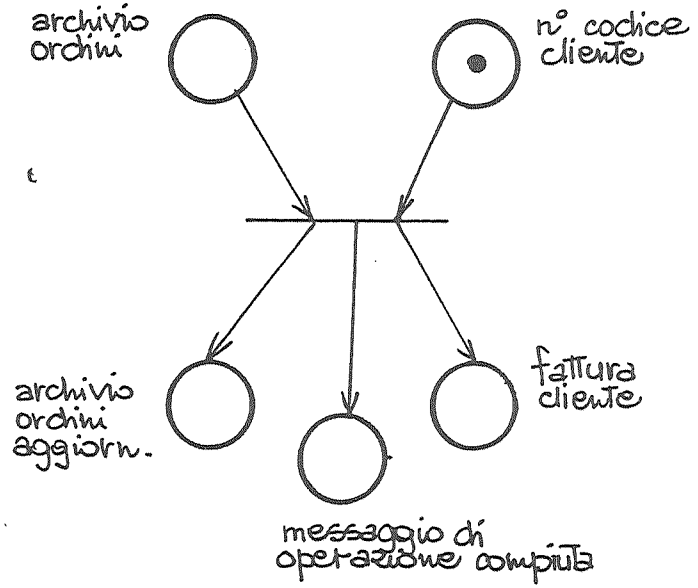
A Assegno da incassare
B Sportello bancario libero
C Conto corrente attivo
D Danaro
E Sportello bancario libero
F Conto corrente ridotto

L'INCASSO (α) DI UN ASSEGNO RICHIEDE LA PRESENZA DI UNO SPORTELLO LIBERO E DI UN C/C ATTIVO, E PRODUCE I SOLDI INCASSATI, RILASCIANDO SPORTELLO E RIDUCENDO C/C.

RETI DI PETRI
EVOLUZIONE

LA PRESENZA DI UNA RISORSA VIENE INDICATA DA UNA "MARCA" INSERITA IN UN POSTO.

Esempio:



STRUMENTI

- **RETI DI PETRI**

  modificate nella forma grafica per una più intuitiva interpretazione, sviluppate top-down

- **TABELLE DI VERIFICA**

  per un controllo delle interazioni sulla base della struttura grafica

- **USO DI PSPN**

  per una descrizione in lingua naturale delle reti disegnate

- **VERIFICHE LINGUISTICHE E COSTRUTTI**

  per specificare le Trasform... in modo non procedurale e per controllare la non-ambiguità delle specifiche

RETI DI PETRI
MODIFICHE

- **RISORSE**

  forma grafica mnemonica

- **ATTIVITÀ**

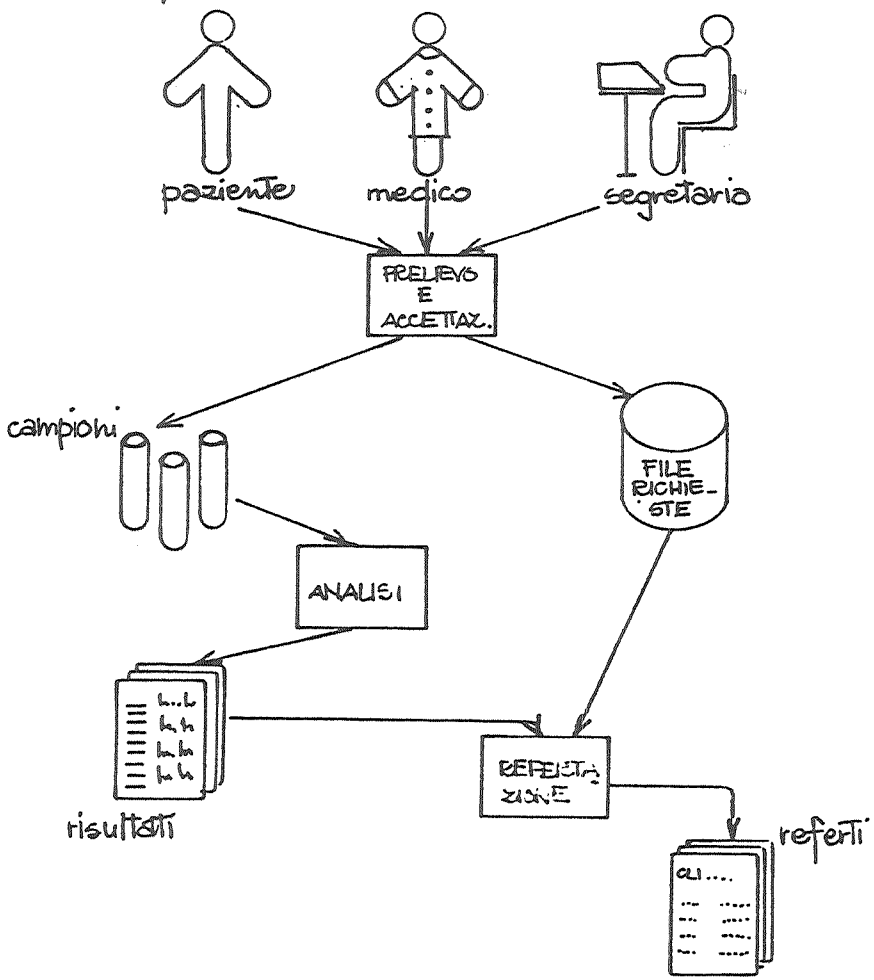
  scatole contenenti descrizione della trasformazione effettuata

- **NOTAZIONI ABBREVIATE**

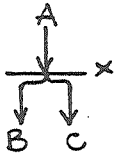
  variazioni grafiche per rappresentare più immediatamente situazioni di concorrenza o conflitto

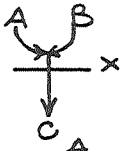
RETI DI PETRI
NOTAZIONI

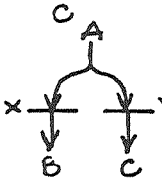
Esempio:

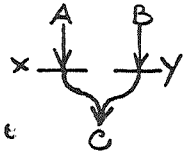


RETI DI PETRI
MODIFICHE

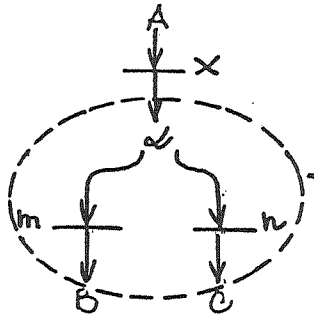
1.  A produce B o (aut) C via X

2.  A o (aut) B producono C via X

3.  A produce B e C via x e y

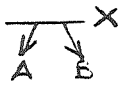
4.  A e B producono C via x e y

NOTAZIONE ESTESA : esempio

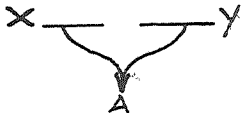


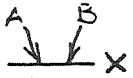
elementi privi di significato in termini di problema

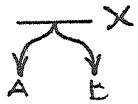
RETI DI PETRI
VERIFICHE

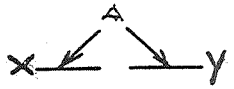
1. Generazione parallela 

2. Alimentazione parallela 

3. Generazione congiunta 

4. Alimentazione congiunta 

5. Generazione alternativa (aut) 

6. Alimentazione alternativa (aut) 

7. Generazione alternativa (vel.) 

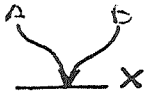
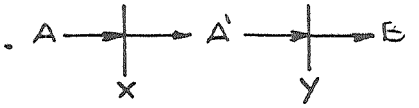
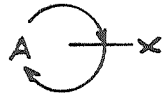
8. Alimentazione alternativa (vel.) 

TABELLA
DI VERIFICA (RETI)

1. 

è fisicamente distinto?

2. 

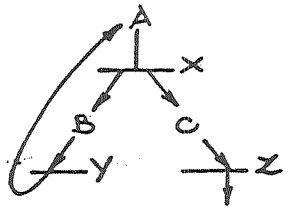
quanti cicli sono permessi?

3. 

concorrenza? interfoliazione?

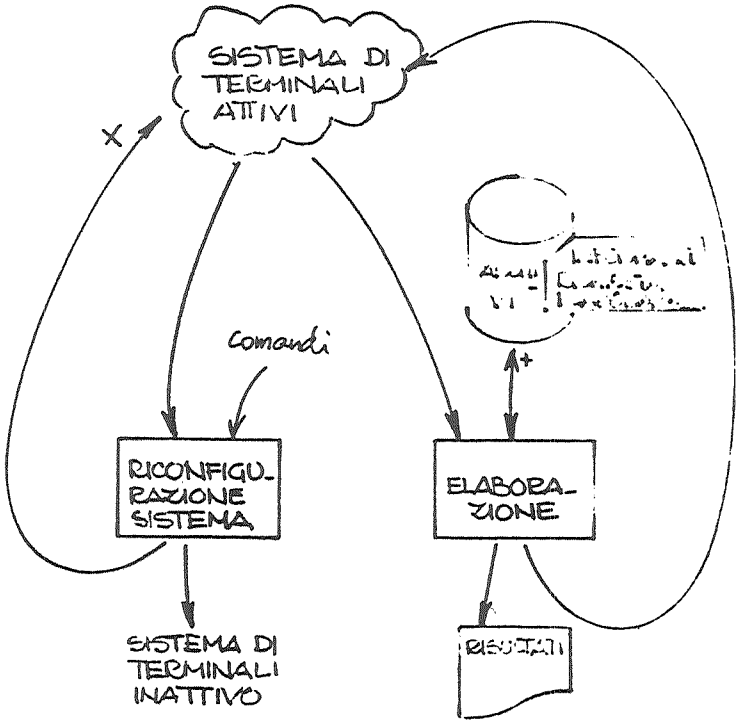
4. 

concorrenza? semafori?

5. 

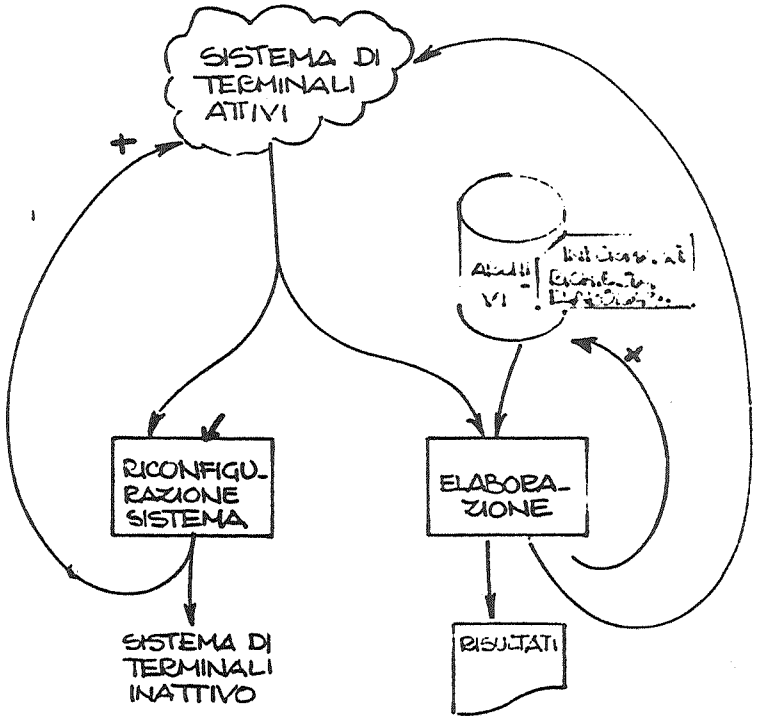
coda su C?

ESEMPIO



VERIFICHE:

ESEMPIO

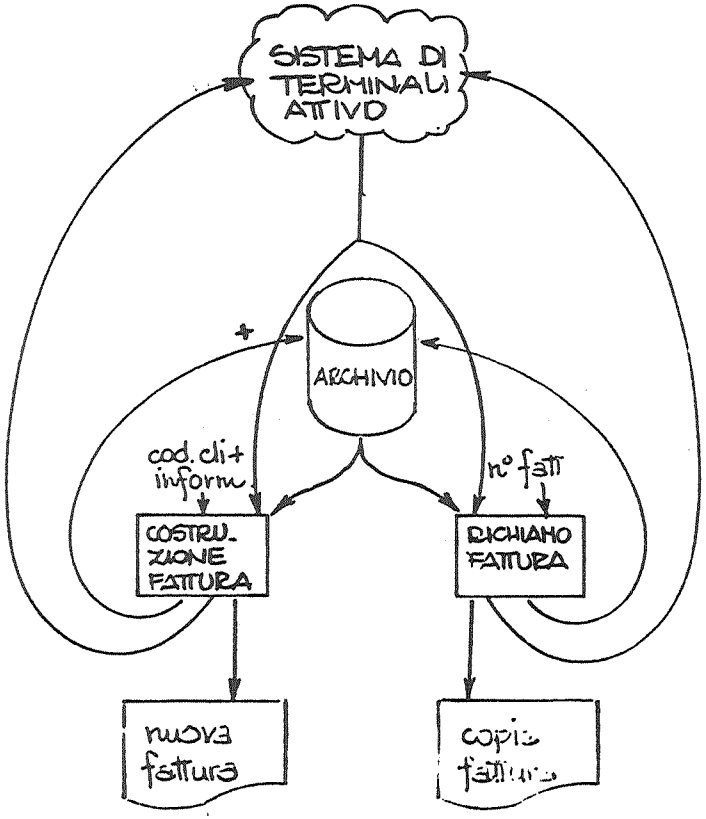


PROCEDURA MODIFICATA

SVILUPPO TOP_DOWN

Raffinamento schema precedente

ELABORAZIONE



PSPN

Il documento di specifica risulta in formato PSPN

- PS descrizione della rete di Petri di livello opportuno
- PN descrizione del comportamento della rete precedente con
  - descrizione di ogni trasformazione
  - descrizione di ogni risorsa
  - Testo indentato

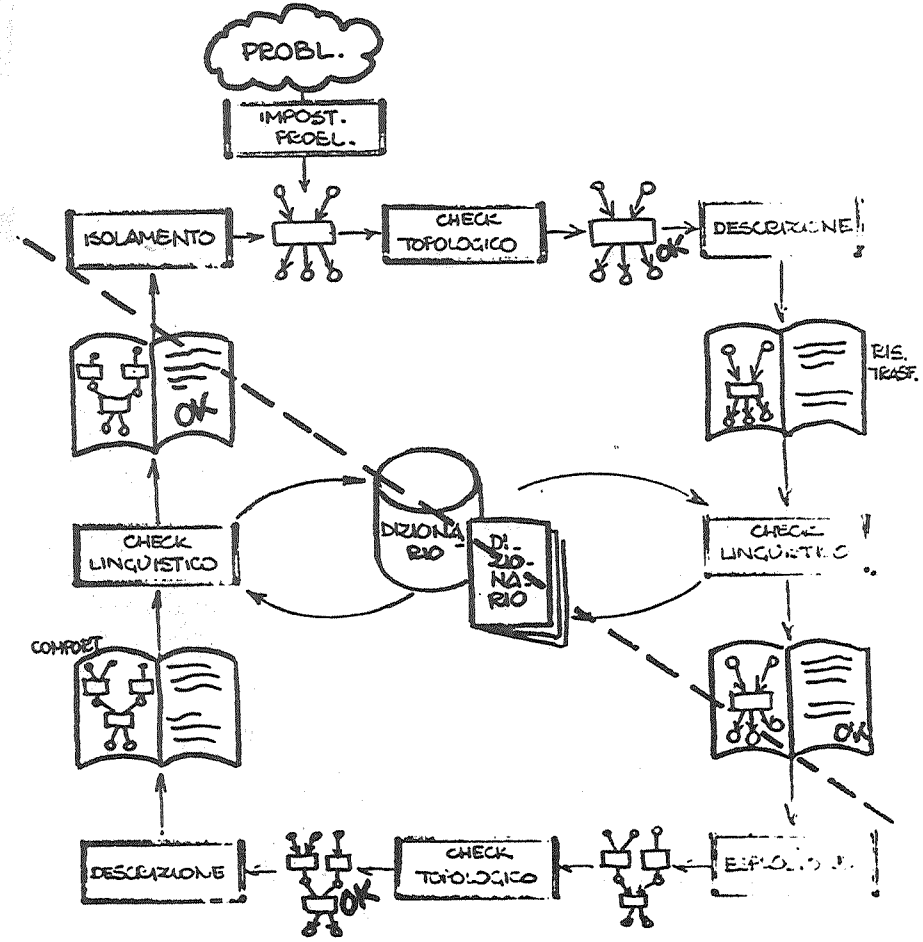
CONTROLLI LINGUISTICI

- **ARTICOLI** controllare: determinativo / indeterminativo / assenza -
- **SOSTANTIVI** ogni plurale o collettivo sia specificato in numero e attributo che qualifica il sottoinsieme di appartenenza -
- **AGGETTIVI/ AVVERBI** devono essere essenziali! Altrimenti eliminarli -
- **VERBI** deve essere chiaro il soggetto e (per i verbi transitivi) il complemento oggetto -
- **PRONOMI** verificare non-ambiguità -
- **CONGIUNZIONI** verificare congiunzioni - Specificare alternative (vel/aut) -

CONTROLLI LINGUISTICI

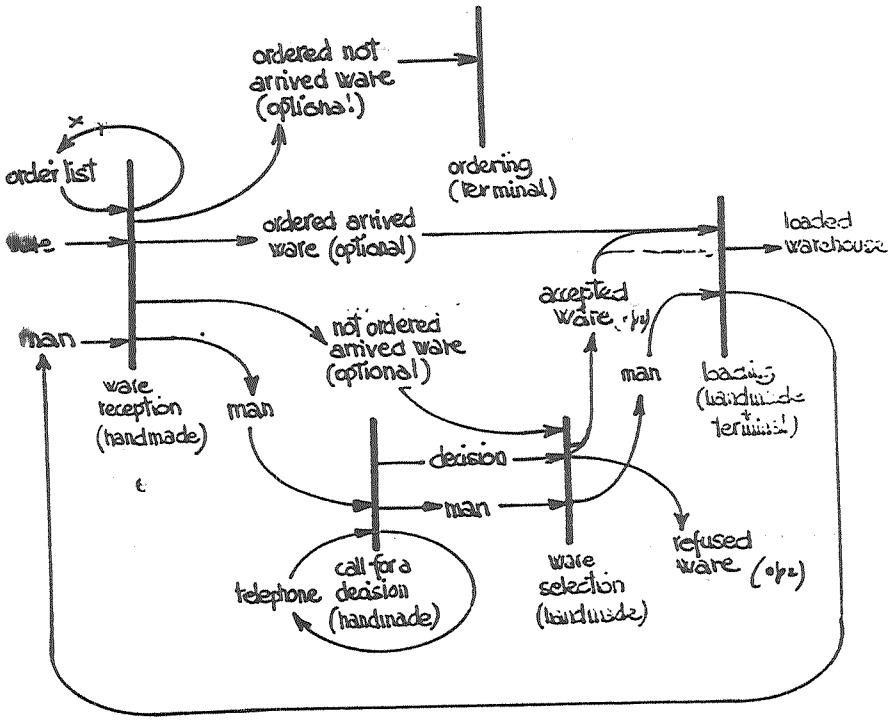
- Un termine non deve essere usato con più significati -
- Un'entità non deve essere descritta con più termini
- Evitare forme impersonali (si legge, ..., si scrive, ...)

SINTESI DELLA METODOLOGIA



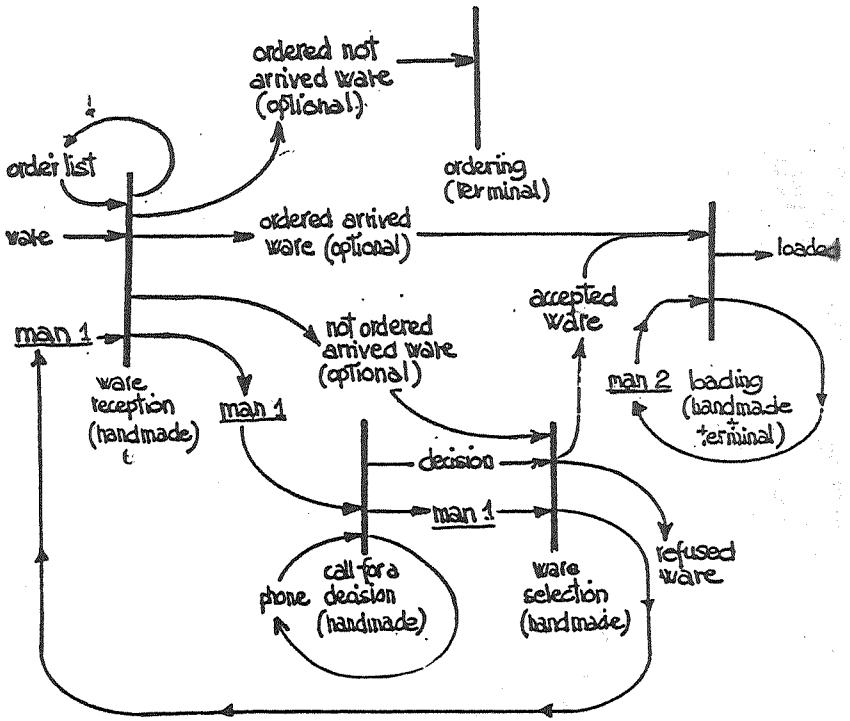
BF 36
(FR 30)

ANALISI ORGANIZZATIVA



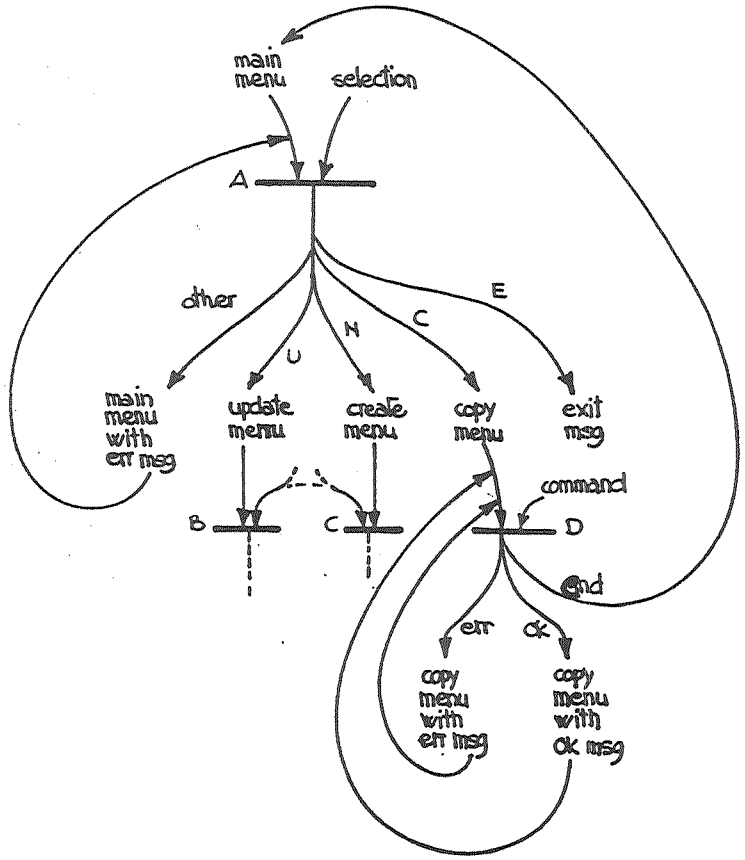
FR 36
(GT 16)

ACTIVITY	MEAN HUMAN TIME SPENT	% OF THE OCCURRENCE	WEIGHTED TIME
Reception	45'	100	45'
Call for a decision	45' + 5'	15	≃ 12'
Ware selection	30'	15	≃ 5'
Loading	2h	100	2h
TOTAL TIME			≃ 3h



	ACTIVITY	MEAN HUMAN TIME SPENT	% OF THE OCCURRENCE	WEIGTED TIME
Man 1	Reception	45'	100	45'
	Call	45' + 5' fix	15	≃ 12'
	Selection	30'	15	≃ 5'
Tot. 1				≃ 1h
Man 2	Loading	2h	100	2h
Tot. 2				2h

RELAZIONI CON ASPETTI DI DISEGNO



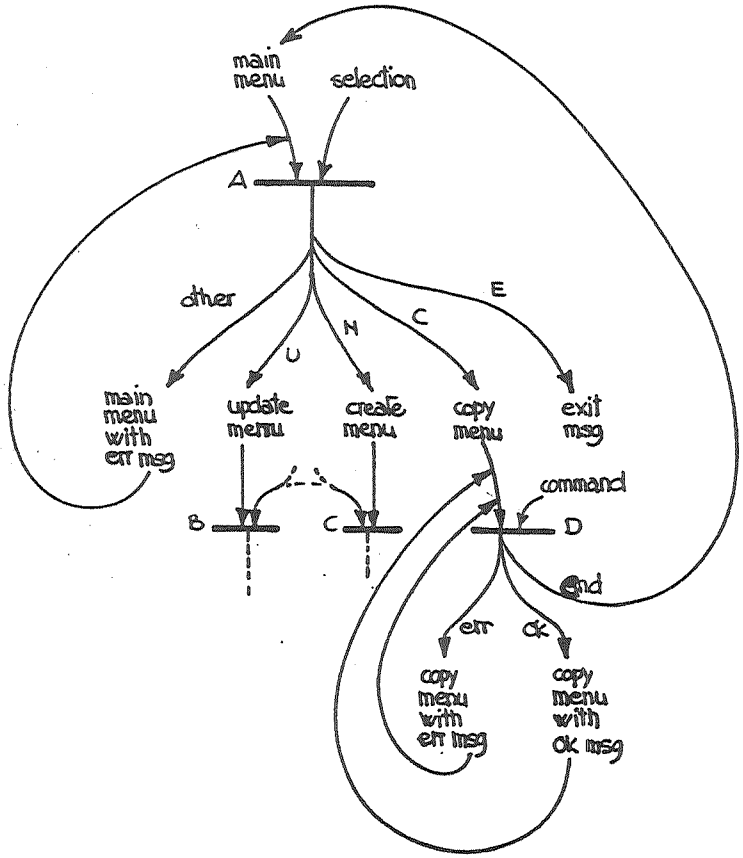
FR43
(GT20)

TRANSACTIONAL ROUTINE	starting mask	events	next routine	output	next mask
A	main menu	E	–	exit msg	–
		C	D	–	copy menu
		N	C	–	create menu
		U	B	–	update menu
		other	A	err. msg	main menu
D	copy menu	end	A	–	main menu
		correct	D	ok msg	copy menu
		errors	D	err msg	copy menu

FR44
(GT21L)

	ACTIVITY	MEAN HUMAN TIME SPENT	% OF THE OCCURRENCE	WEIGTED TIME
Man 1	Reception	45'	100	45'
	Call	45' + 5' fix	15	≃ 12'
	Selection	30'	15	≃ 5'
Tot. 1				≃ 1h
Man 2	Loading	2h	100	2h
Tot. 2				2h

RELAZIONI CON ASPETTI DI DISEGNO



FR43
(GT20)

TRANSACTIONAL ROUTINE	starting mask	events	next routine	output	next mask
A	main menu	E	–	exit msg	–
		C	D	–	copy menu
		N	C	–	create menu
		U	B	–	update menu
		other	A	err. msg	main menu
D	copy menu	end	A	–	main menu
		correct	D	OK msg	copy menu
		errors	D	err msg	copy menu

FR44
(GT21L)

DAFNE®

DATA

FUNCTIONS

NETWORKING

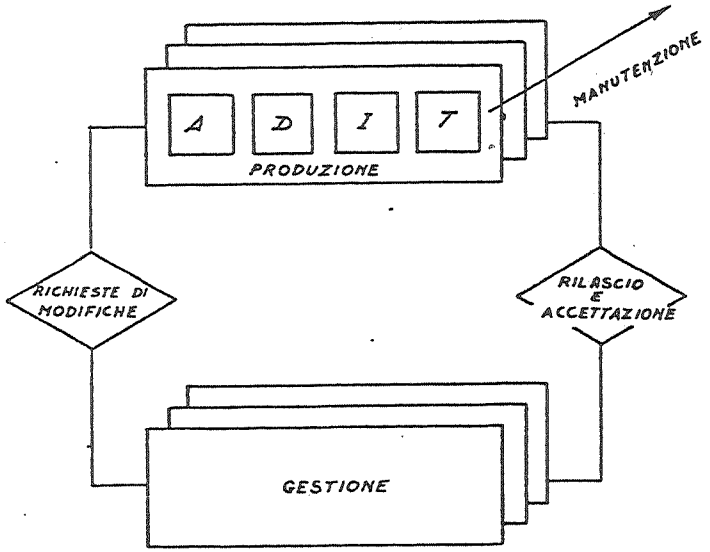
Doh.ssa LINDA CRIMERSMOIS

QUADRO METODOLOGICO PER L'ANALISI E IL DISEGNO DEI SISTEMI SOFTWARE

I·N·D·I·C·E

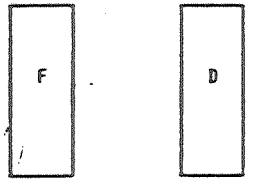
PROPRIETA' DELLA ITALSIEL S p A

REQUISITI: UNA METODOLOGIA PER LO SVILUPPO E LA MANUTENZIONE DEL SOFTWARE

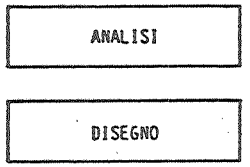


"APPROCCIO STRUTTURATO" ALLE FASI ALTE

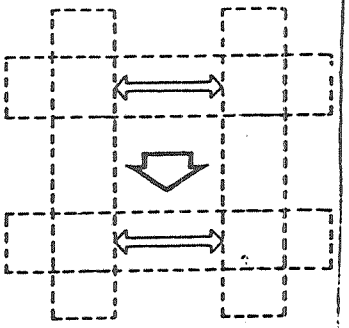
SEPARAZIONE DATI/FUNZIONI



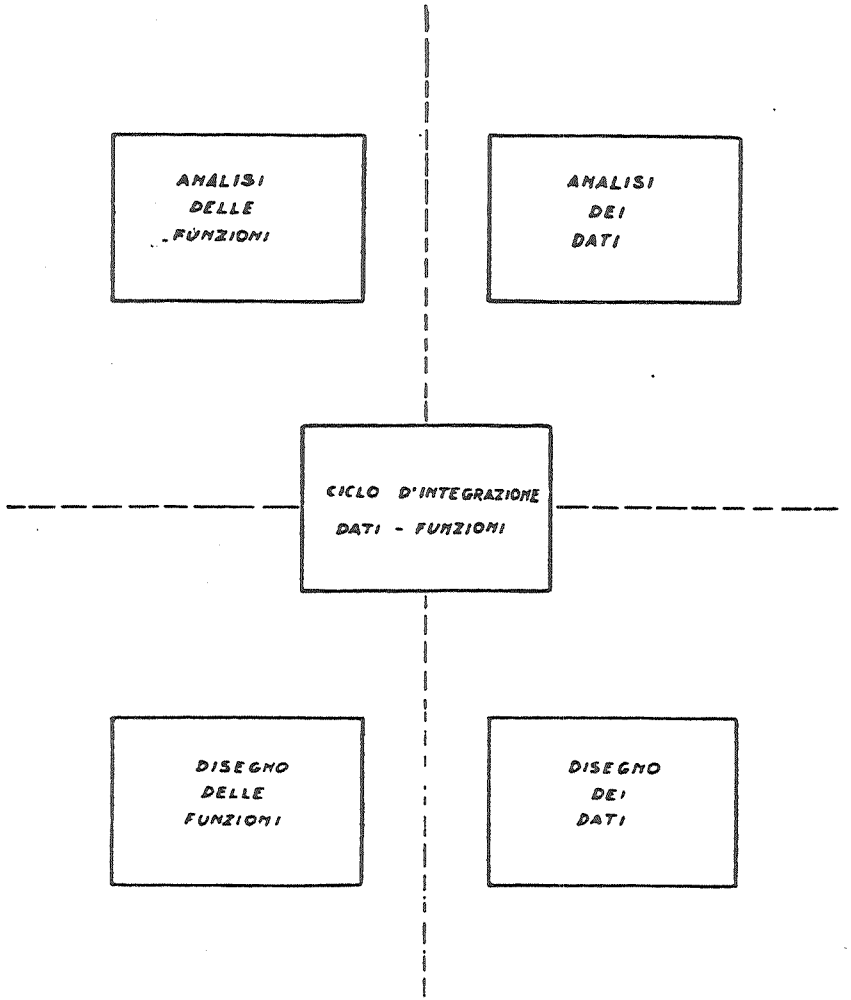
SEPARAZIONE ANALISI/DISEGNO



GESTIONE DELLE INTERFACCE



DAFNE : DATA AND FUNCTIONS NETWORKING



UN QUADRO ORIGINALE PER GESTIRE IL PROCESSO DI PRODUZIONE

DI SISTEMI APPLICATIVI

... CARATTERISTICHE

LE CARATTERISTICHE SALIENTI DI DAFNE SONO

- MODULARITÀ

- DIVERSI LIVELLI DI MODELLIZZAZIONE IN ANALISI
  - . LIVELLO INDIPENDENTE DALLE SCELTE DI AUTOMAZIONE
  - . LIVELLO INDIPENDENTE DALLE SCELTE DI REALIZZAZIONE

- CONTEMPORANEITÀ DELLE ATTIVITÀ DI SPECIFICAZIONE E DOCUMENTAZIONE

- STRUTTURAZIONE DEL SOFTWARE SUI DATI

SEPARAZIONE DELLE TECNICHE E DEI LINGUAGGI

PER ASSICURARE L'INDIPENDENZA DELLE ATTIVITÀ È NECESSARIO ADOPERARE TECNICHE DI ANALISI E SIMBOLISMI DIVERSI

- LE DUE ATTIVITÀ SONO IN SEQUENZA: UN CAMBIAMENTO DI LINGUAGGIO PERMETTE DI CONTROLLARE LA LORO SEPARAZIONE

ANALISI

DISEGNO

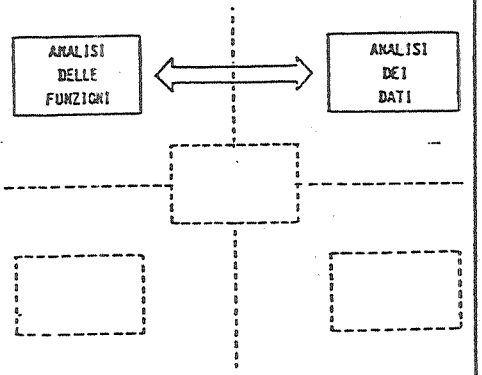
- LE DUE ATTIVITÀ SONO PARALLELE: L'USO DI TECNICHE SPECIFICHE È INDISPENSABILE PER ASSICURARE LA LORO SEPARAZIONE

FUNZIONI | DATI

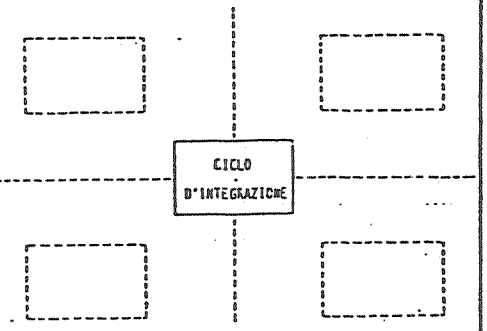
LE INTERFACCE TRA LE ATTIVITA'

L'INDIPENDENZA DELLE ATTIVITÀ E DELLE TECNICHE UTILIZZATE OFFRE APPROPRIATI STRUMENTI:

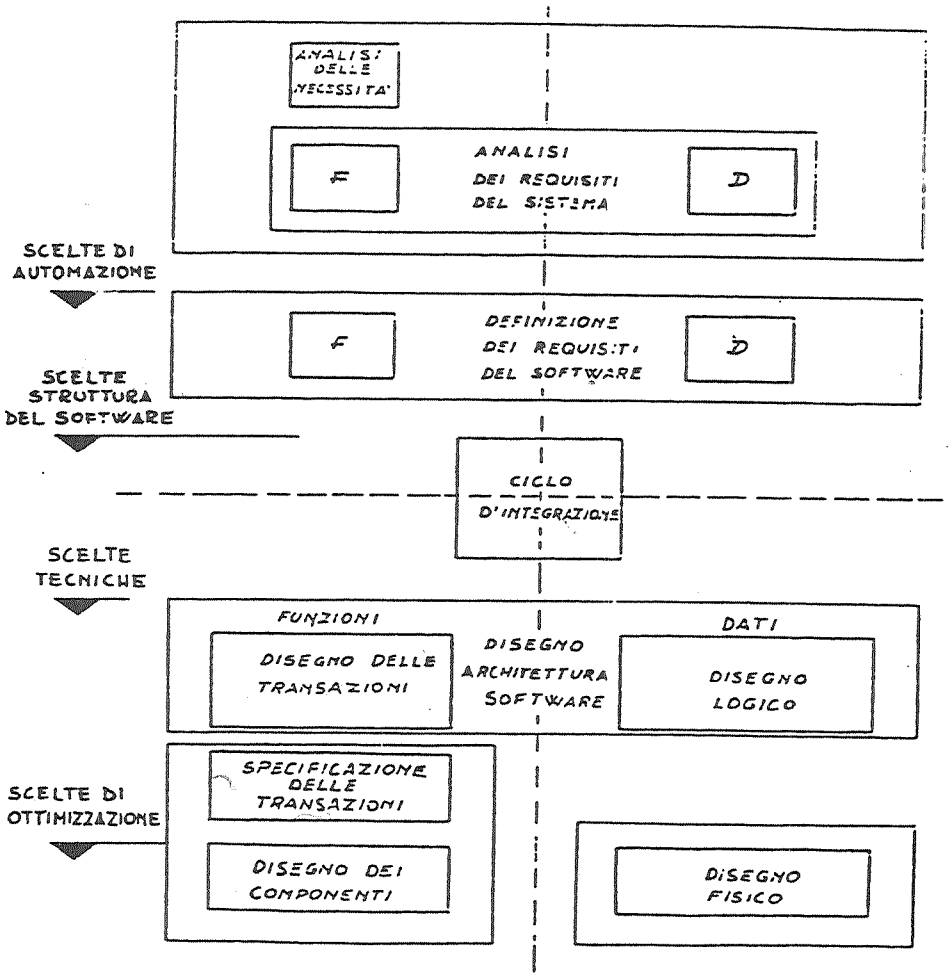
- PER DEFINIRE I PUNTI DI ARRESTO E I CRITERI DI VERIFICA DELLA COMPLETEZZA DELL'ANALISI



- PER LOCALIZZARE IL PASSAGGIO ANALISI - DISEGNO IN UNA SPECIFICA FASE METODOLOGICA CHE SI ATTUA ATTRAVERSO IL CONFRONTO FUNZIONI - DATI.



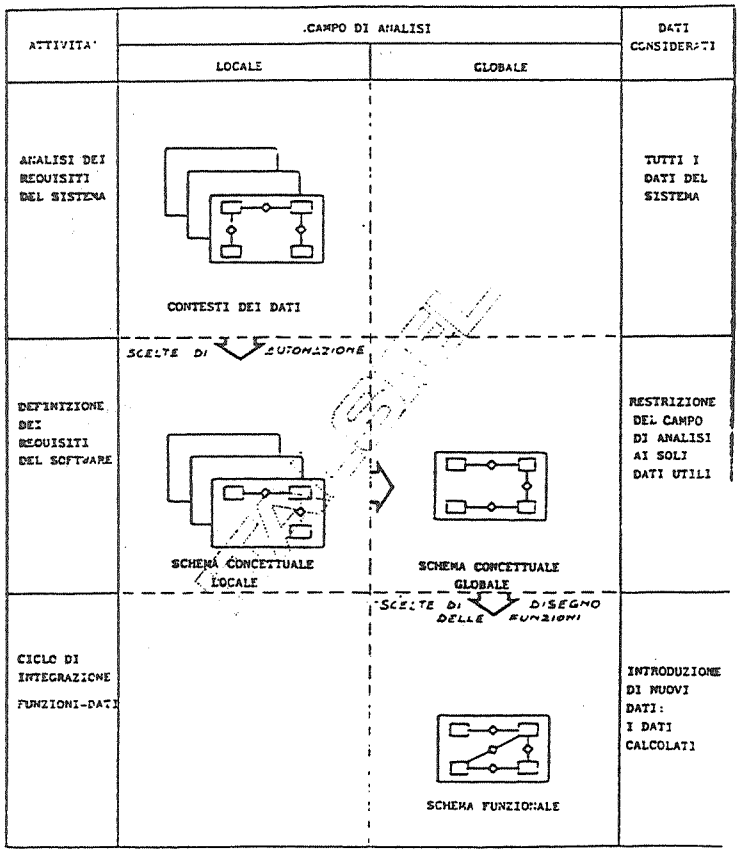
I LIVELLI DI ASTRAZIONE RISPETTO AI VINCOLI REALIZZATIVI IN DAFNE



I PRODOTTI DELL'ANALISI DELLE FUNZIONI

ATTIVITA'	PRODOTTI		
ANALISI DEI REQUISITI DEL SISTEMA	MODELLI FUNZIONALI	MODELLO GLOBALE DEL NUOVO SISTEMA	IL CONTESTO E' IL SISTEMA INFORMATIVO
DEFINIZIONE DEI REQUISITI DEL SOFTWARE	SCELTE DI AUTOMAZIONE MODELLI DEI REQUISITI DEL SOFTWARE	MODELLO GLOBALE DEI REQUISITI DEL SOFTWARE	RESTRIZIONI DEL CAMPO DI ANALISI ALLA PARTE AUTOMATIZZABILE DEL SOFTWARE
IL CICLO D'INTEGRAZIONE FUNZIONI-DATI	SCELTE SULLA STRUTTURA DEL SOFTWARE DESCRIZIONE ANALITICA DI TRANSAZIONE		TRADUZIONE DEI RISULTATI DELL'ANALISI SOTTO FORMA OPERATIVA: IDENTIFICAZIONE DELLE TRANSAZIONI

I PRODOTTI DELL'ANALISI DEI DATI



CARATTERISTICHE DEL SOFTWARE

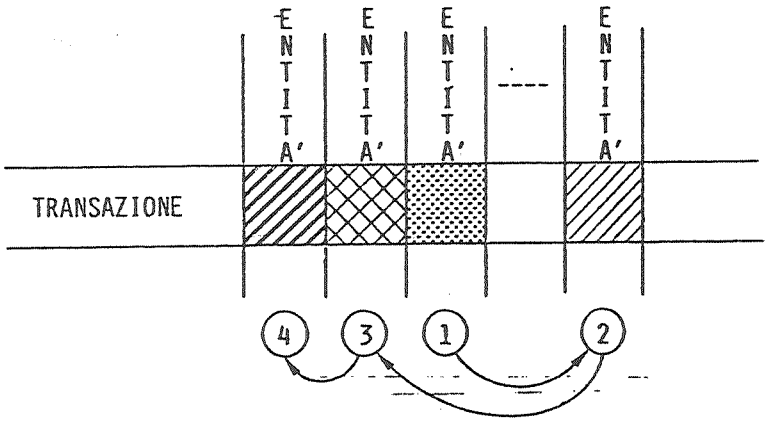
IL RISULTATO DELL'APPLICAZIONE DI DAFNE ALLA PROGETTAZIONE DI UN SISTEMA APPLICATIVO È UN SOFTWARE MODULARE, STRUTTURATO SUI DATI CHE PRESENTA CARATTERISTICHE SPINTE DI

- INFORMATION HIDING : SODDISFATTA PER MEZZO DELLA SEPARAZIONE TRA COMPONENTI DI CONTROLLO E ATTIVAZIONE E COMPONENTI APPLICATIVI DI TRATTAMENTO DEI DATI

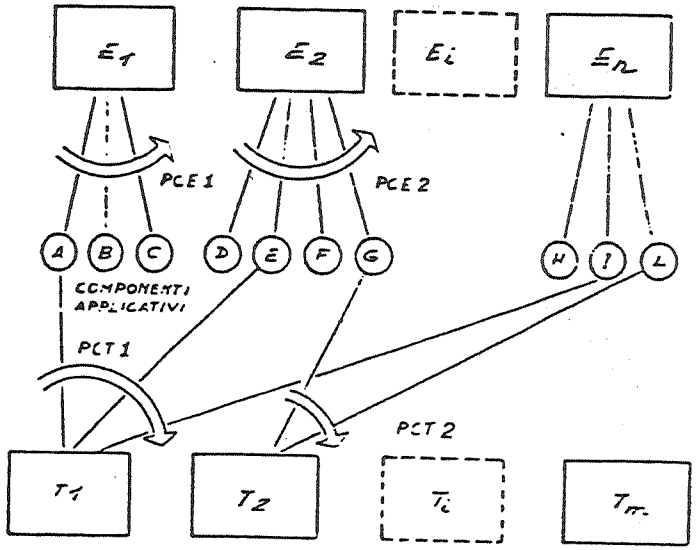
- LOCALIZZAZIONE : I COMPONENTI APPLICATIVI TRATTANO PORZIONI LOGICAMENTE LIMITATE DEI DATI DELL'APPLICAZIONE

... Caratteristiche

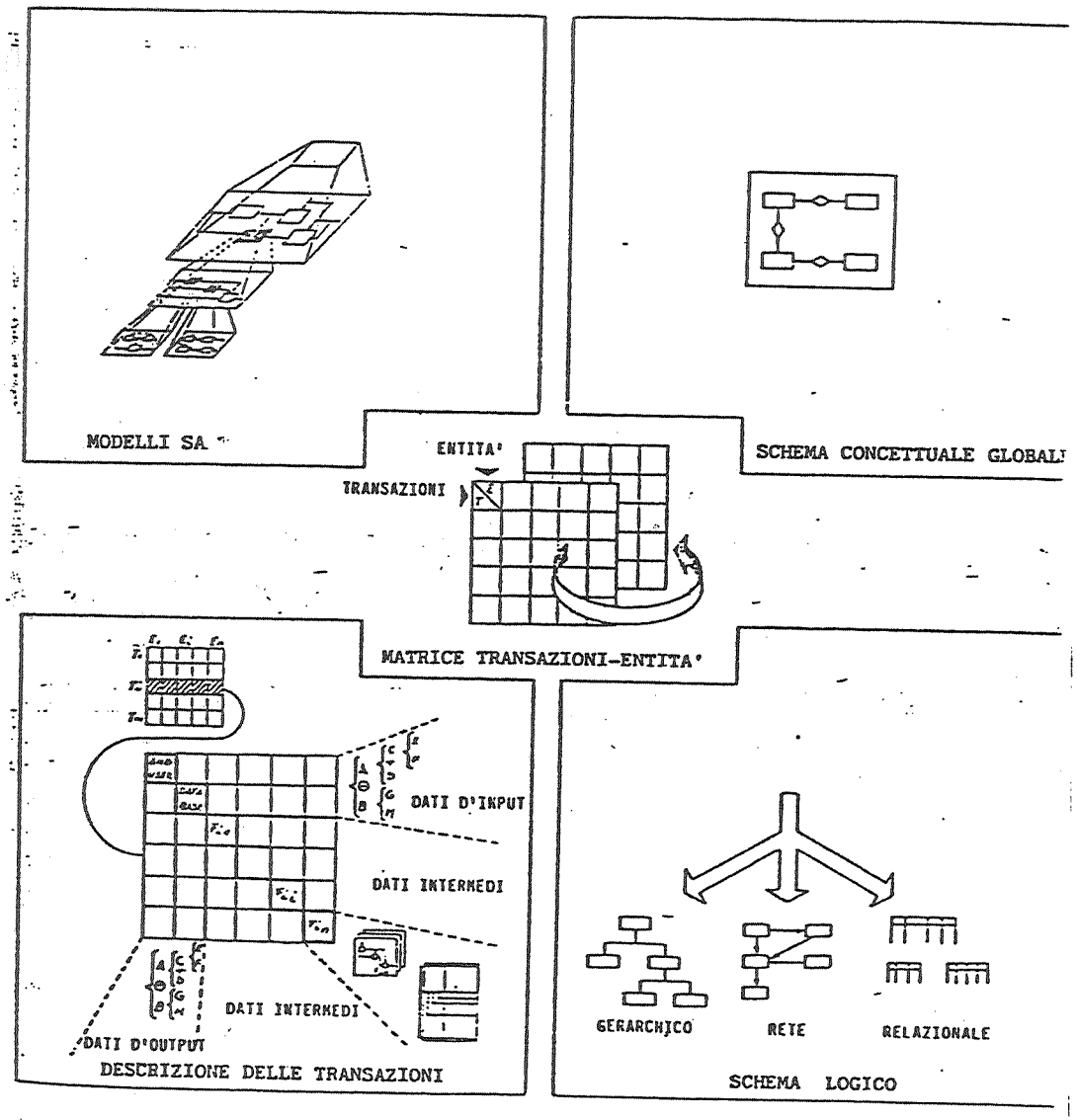
Con DAFNE si perviene a una situazione in cui una transazione si articola in una sequenza di componenti applicativi indipendenti tra loro la cui attivazione è guidata da un componente di controllo e attivazione



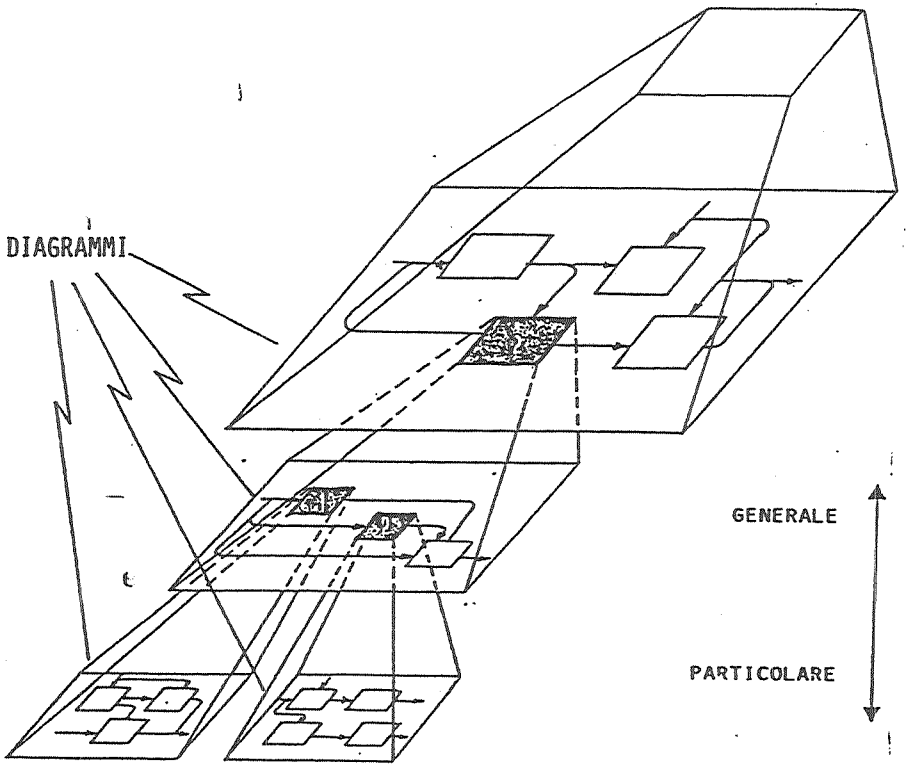
ARCHITETTURA DEL SOFTWARE



TECNICHE DI SPECIFICAZIONE IN DAFNE



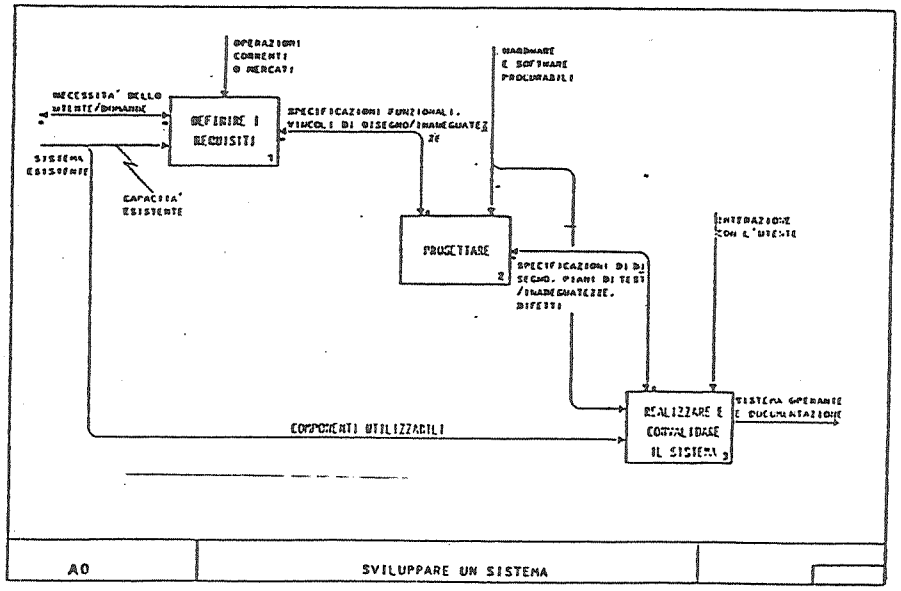
LA SCELTA ITALSIEL PER L'ANALISI DELLE FUNZIONI: SA



. UN MODELLO SA E' UNA SEQUENZA ORGANIZZATA DI DIAGRAMMI CHE RAPPRESENTA LA REALTÀ ATTRAVERSO UNA DESCRIZIONE GERARCHICA E MODULARE

ELEMENTI DI UN DIAGRAMMA SA

UN DIAGRAMMA È COSTITUITO DA BLOCCHI CONNESSI DA FRECCE CHE DESCRIVONO VINCOLI ED INTERFACCE

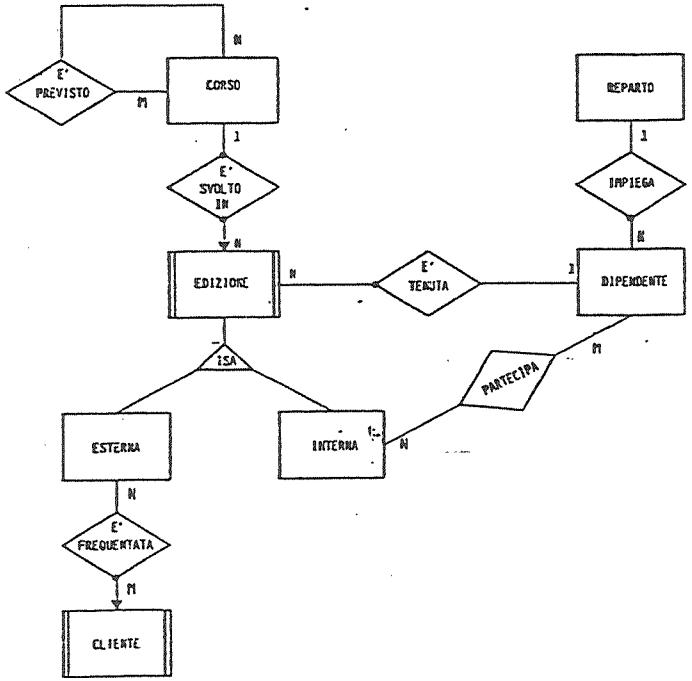


PRINCIPALI RISULTATI CONSEGUIBILI CON SA

- ORGANIZZARE L'ANALISI PER MEZZO DEI CONCETTI DI
  - . SCOPO : RISULTATO CHE SI VUOLE CONSEGUIRE CON IL SINGOLO MODELLO
  - . CONTESTO : FRONTIERA DEL PROBLEMA
  - . PUNTO DI VISTA: CARATTERISTICHE E LIVELLO DI DETTAGLIO DELLA RAPPRESENTAZIONE
- DOCUMENTARE IL SISTEMA DURANTE L'ANALISI E NON DOPO AVERLO ANALIZZATO
- DIFFONDERE I RISULTATI DELL'ANALISI
  - . COMUNICAZIONE (INTERCAMBIABILITÀ ...)
  - . CONTROLLO DI QUALITÀ (CICLO DI REVISIONE)
- ASSICURARE IL CONTROLLO DI AVANZAMENTO DEL LAVORO

INGEGNERIZZAZIONE E INTEGRAZIONE DELL'ENTITY --RELATIONSHIP APPROACH

LA SCELTA ITALSIEL PER L'ANALISI DEI DATI: L'ENTITY RELATIONSHIP APPROACH (ERA)



LA TECNICA DI RAPPRESENTAZIONE DEI DATI E' "PIATTA", NON GERARCHICAMENTE STRUTTURATA COME QUELLA DELLE FUNZIONI

CONCETTI BASE

ENTITÀ — OGGETTO O CONCETTO CHE PUÒ ESSERE IDENTIFICATO DISTINTAMENTE

RELAZIONE — "LEGAME" TRA PIÙ ENTITÀ AL QUALE È ASSOCIATO UN SIGNIFICATO

ATTRIBUTO — PROPRIETÀ CHE CARATTERIZZA UNA ENTITÀ O UNA RELAZIONE

. . . . . .

L'INGEGNERIZZAZIONE DELL'ENTITY RELATIONSHIP APPROACH DI P.P.S. CHEN CONSISTE IN:

. DEFINIZIONE DEL PROCEDIMENTO

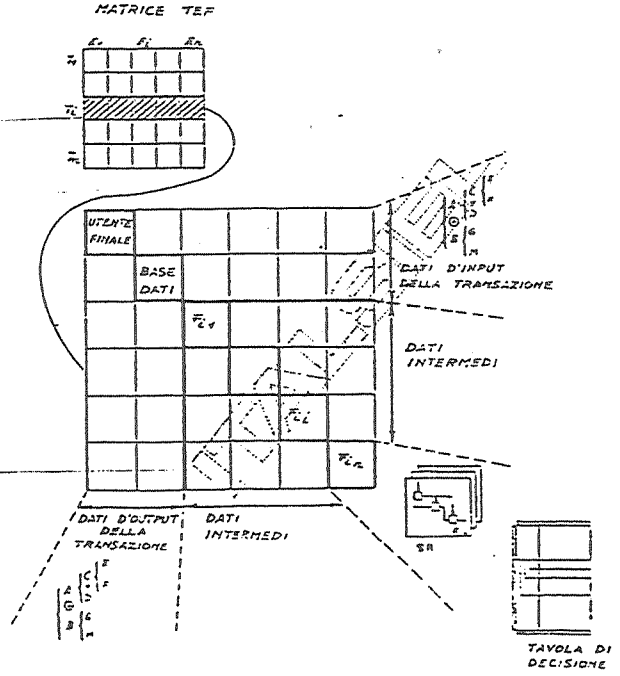
  - SUDDIVISIONE ORIZZONTALE DEL CAMPO DI ANALISI IN LIVELLI
  - SUDDIVISIONE VERTICALE DEL CAMPO DI ANALISI IN VISTE LOCALI

. ARRICCHIMENTO DEL MODELLO ENTITY-RELATIONSHIP

SPECIFICAZIONE DELLE TRANSAZIONI

OGNI TRANSAZIONE VIENE SPECIFICATA PER MEZZO DI UN KIT DI TRANSAZIONE COSTITUITO DA

- UN DIAGRAMMA N^2
- UNA DESCRIZIONE DELL'INPUT/OUTPUT (WARNIER)
- REGOLE DI ATTIVAZIONE DELLE FUNZIONI (TAVOLE DI VERITÀ E WARNIER)
- DESCRIZIONE DETTAGLIATA DELLE FUNZIONI
- INFORMAZIONI PER LA REALIZZAZIONE

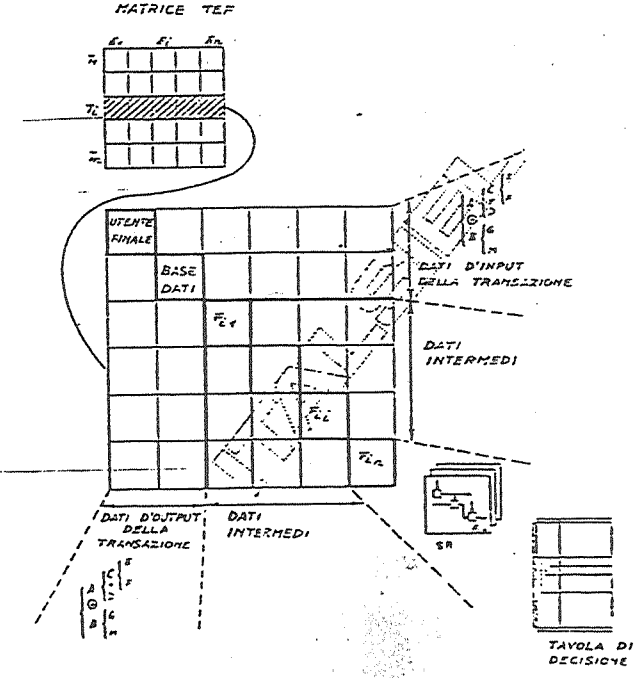


... TECNICHE

SPECIFICAZIONE DELLE TRANSAZIONI

OGNI TRANSAZIONE VIENE SPECIFICATA PER MEZZO DI UN KIT DI TRANSAZIONE COSTITUITO DA

- UN DIAGRAMMA N^2
- UNA DESCRIZIONE DELL'INPUT/OUTPUT (WARNIER)
- REGOLE DI ATTIVAZIONE DELLE FUNZIONI (TAVOLE DI VERITÀ E WARNIER)
- DESCRIZIONE DETTAGLIATA DELLE FUNZIONI
- INFORMAZIONI PER LA REALIZZAZIONE



PROPRIETA' DELLA ITALSIEL S p A

IL DIAGRAMMA N^2

- LE FUNZIONI SONO SCRITTE SULLA DIAGONALE
- GLI OUTPUT SONO SCRITTI PER RIGHE
- GLI INPUT SONO SCRITTI PER COLONNE
- LE CASELLE CHE NON SI TROVANO SULLA DIAGONALE INDICANO LE INTERFACCE TRA LE FUNZIONI

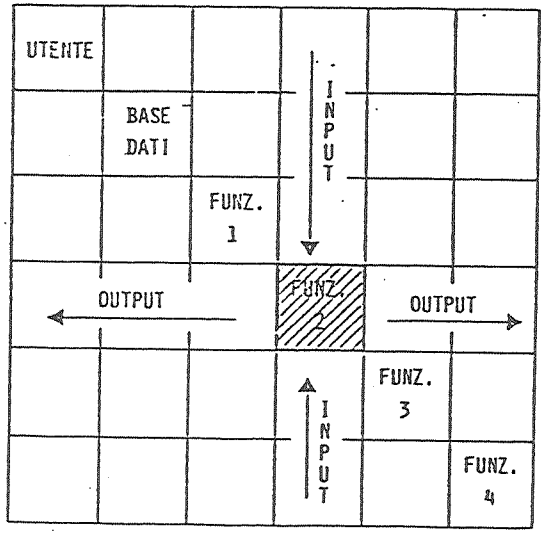


DIAGRAMMA N^2

GESTIONE DELLA COMPLESSITA'

I METODI DI ANALISI TOP-DOWN RICHIEDONO:

- UN UNICO PUNTO DI PARTENZA PER L'ATTIVITÀ
- STRUTTURA DEL PROBLEMA FACILMENTE RICONDUCIBILE AD UNO SCHEMA GERARCHICO

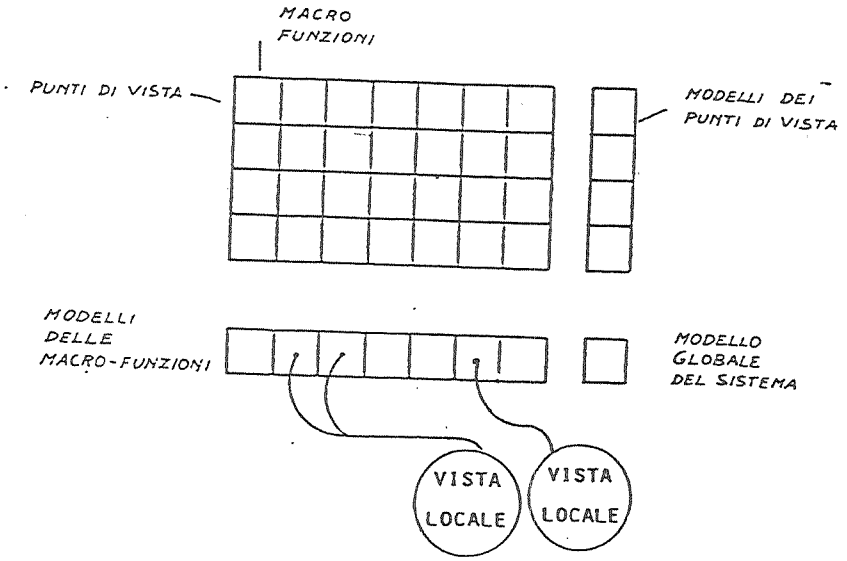
NON SEMPRE CIÒ È POSSIBILE, IN PARTICOLARE AGLI ALTI LIVELLI DI ANALISI.

PER AFFRONTARE LA COMPLESSITÀ IN ANALISI, DAFNE PROPONE DI UTILIZZARE SPECIFICI MODELLI SAS PER

- DESCRIVERE L'AMBIENTE
- SUDDIVIDERE IL CAMPO DI ANALISI
- DESCRIVERE LE INTERAZIONI TRA LE PARTI

LO STRUMENTO PER GESTIRE LA COMPLESSITÀ IN ANALISI È LA MATRICE DEI MODELLI CHE PERMETTE DI

- IDENTIFICARE I MODELLI SA DA SVILUPPARE
- SUDDIVIDERE IL CAMPO DELL'ANALISI DEI DATI IN VISTE LOCALI



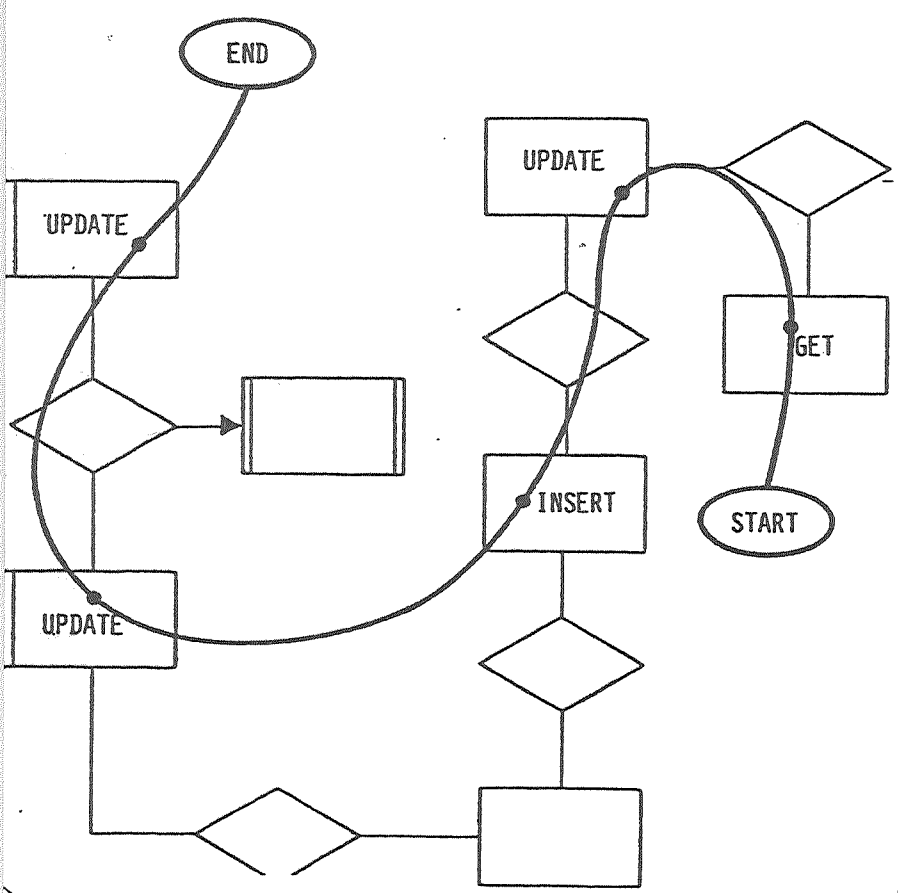
UN METODO ORIGINALE PER IL PASSAGGIO DELL'ANALISI AL DISEGNO

IL CICLO D'INTEGRAZIONE DATI-FUNZIONI E' IL "PONTE" TRA ANALISI E DISEGNO E SI SVOLGE:

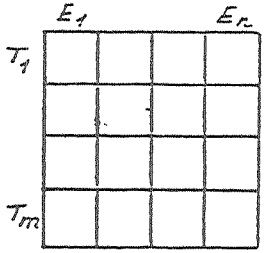
- SVILUPPANDO UN CICLO IPOTESI-VALIDAZIONE CHE DETERMINA SIMULTANEAMENTE LO SCHEMA DEI DATI E LE TRANSAZIONI DA REALIZZARE
- DOCUMENTANDO LE SCELTE

ANALISI DELLE INTERAZIONI TRANSAZIONI-DATI

- OGNI TRANSAZIONE È VISTA COME UN INSIEME DI CAMMINI E DI TIPI DI FUNZIONI (GET, INSERT, UPDATE, DELETE) CHE OPERANO SULLO SCHEMA FUNZIONALE



- L'INTERAZIONE TRA L'INSIEME DELLE TRANSAZIONI E LO SCHEMA FUNZIONALE È CONTROLLATA DALLA "MATRICE T E" (TRANSAZIONI-ENTITÀ)



SULLA QUALE

- OGNI COLONNA SI RIFERISCE AD UN'ENTITÀ (O RELAZIONE CON ATTRIBUTI) DELLO SCHEMA FUNZIONALE

- OGNI RIGA SI RIFERISCE AD UNA TRANSAZIONE

- OGNI ELEMENTO EVIDENZIA IL TIPO DI FUNZIONE (G,I,U,D) CHE DEVE ESSERE COMPIUTA SU UN'ENTITÀ DA UNA TRANSAZIONE E IL NUMERO DI OCCORRENZE COINVOLTE

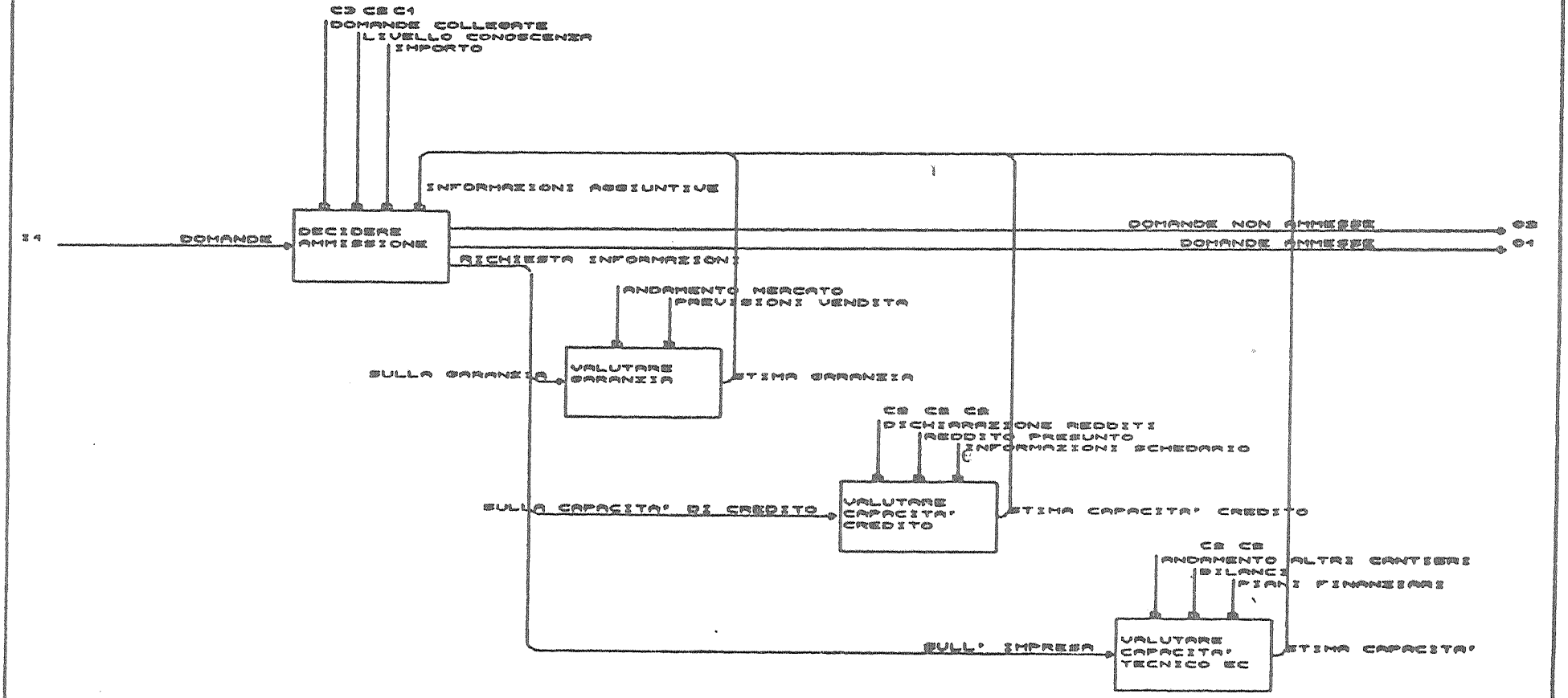
OBIETTIVI

- SUPPORTO ALLA CONDUZIONE TECNICA DI UN PROGETTO
  - . LIBRERIA DEI PRODOTTI
  - . DATA-BASE DI PROGETTO
- SUPPORTO AL PROGETTISTA APPLICATIVO NELLA CONDUZIONE DELLE ATTIVITA'
  - . SVILUPPO DEI PRODOTTI
  - . VERIFICA DELLA CORRETTEZZA DI OGNI SINGOLO PRODOTTO
  - . VERIFICA DELLA COERENZA DI UN INSIEME DI PRODOTTI
  - . PRODUZIONE E GESTIONE DELLA DOCUMENTAZIONE

SETTORI DI INTERVENTO

- ANALISI FUNZIONI
  - . CREAZIONE E GESTIONE DI MODELLI E DIAGRAMMI SA
  - . GESTIONE DELLE VERSIONI DEI MODELLI
  - . OTTIMIZZAZIONE E TRACCIAMENTO DEI DIAGRAMMI
- ANALISI DATI
  - . CREAZIONE E GESTIONE DEI MODELLI ERA E DELLA RELATIVA DOCUMENTAZIONE
- DISEGNO FUNZIONI

USED AT'	AUTHOR'	DATE' 01 SEP 84	X	WORKING'	READER DATE	CONTEXT
	PROJECT'	REV'		DRAFT		
				RECOMMENDED		
	NOTES ' 1 2 3 4 5 6 7 8 9 10			PUBLICATION		



NODE'	TITLE'	NUMBER
A0	ESEGUIRE ISTRUTTORIA	IFIG 4

COHERENT DEVELOPMENT METHODS IN AN INDUSTRIAL HIGH LEVEL LANGAGE ENVIRONMENT *

K. Ripken
Division Informatique
Laboratoires de Marcoussis
C.G.E. Research Center
F - 91460 MARCOUSSIS

* This paper is published in : NATO ASI Series, Vol. F8, Program Transformation and Programming Environments, Edited by P. PEPPER, Springer-Verlag Berlin Heidelberg 1984, pp. 85-96.

INTRODUCTION

Most of the material presented here are results of a study entitled "Life Cycle Support in the Ada Environment" which was sponsored by the CEC and conducted by Systems Designers Limited (Great Britain) and Tecsi Software (France) in 1982 (SyT 83, McR 84).

The study investigated methods and tools for supporting the development and maintenance of large scale software systems written in Ada. In particular, the study emphasized the integration of the methods and tools for the whole range of software life cycle activities : software development (through the various stages from requirements expression to module coding), integration, maintenance, configuration control and project management. The first part of the presentation will therefore briefly point out the objective of the study, the second part will introduce its software development process model, and in the third part the development methods which have been investigated will be discussed and the possibilities for tools support will be pointed out before a few remarks will conclude the presentation.

In fact the title of this presentation gives the necessary clues to the understanding of the objectives of the study and indicates its emphasis and hypotheses.

The coherency of methods is a conditio sine qua non for the automation of the production of software, for the estimation of the efforts going into the production, the estimation of impacts of changes during the development, and also for the verification and validation of the various software system representations. The plural in "development methods" is to say that we believe that there is not just one method existing which tells how to develop software. In fact, in the industrial situation several teams of people with different skills are needed, decisions have to be made in different phases of the development, and therefore different methods and formalisms are useful in different situations. Methods are also evolving due to progress computer science and to accumulation of experience. However, all the various methods should be governed by one overall method which provides a coherent framework. Talking about an industrial environment means talking about demanding projects where an important volume of software with a life cycle of about twenty to fourty years is developed by more than 100 people, possibly at several sites. On such projects, the requirements may change in the course of the development due to changing political and economical constraints.

As for the high level langage environment, we take the Ada environment in this presentation as the example. Most of the remarks made below apply equally well to environments based on comparable high level languages. The development of the Ada language has provided the important opportunity to place considerable emphasis on integrated environments to support all aspects of the development and maintenance of large scale software systems written in Ada. The reason is that the perspective of Ada as a common high-level language in world-wide use renders for the first time, economically and practically, feasible the development of such environments which are so badly needed in order to improve productivity, quality and reliability of software production.

The requirements for these environments were first described in the "Stoneman" document (Sto 80). Stoneman introduced the idea of a Kernel of an Ada Programming Support Environment (APSE), called KAPSE, which constitutes a portability interface between the APSE tools and the system which hosts the APSE. Furthermore, Stoneman proposed a phased bottom-up approach to the development of APSEs starting with the KAPSE and a minimal set of tools, i.e. with a minimal APSE, called MAPSE. Projects to develop MAPSE level environments are in progress in both Europe and the USA.

Whereas Stoneman is primarily concerned with the MAPSE and says little about the requirements for a full APSE, the study by Systems Designers Ltd. and Tecsi considered the APSE as a whole in a top-down approach. It was to perform a first quick iteration of the design of an APSE. This iteration could, of course, only be preliminary, provide a basis for discussion, and indicate areas of further study, enhancement, or adaptation to Ada.

The emphasis was on demanding industrial projects and on feasibility. Good existing methods were to be combined in a flexible way, practical techniques were to be used, and adaptation was to be done when necessary. The study was to achieve to be a kind of seminal document which points out the feasibility, the benefits and the problems of a programming support environment development, which provides a basis for the classification of terminology, concepts, and objectives, and which identifies further work.

About at the same time, a study called "Methodman" was performed in the USA which addressed the same problem area (Met 82). It stated requirements and performed a survey of existing methodologies. The requirements established match very well with the one of the European study, and the survey led to results which were assumed at the outset of the European study. These results are that there is a white range of available methodologies also called support environments but that each alone is rather incomplete, very often closed, and that it has poor

team support ; that most of them are new, in little wide-spread use, and in a research stage ; that a combination of them is generally incompatible and also incompatible with Ada ; that issues of tools, management and training are not well addressed ; in fact, that the range covered by the methodologies is rather small and that integration is necessary - if at all possible - to cover the whole life cycle.

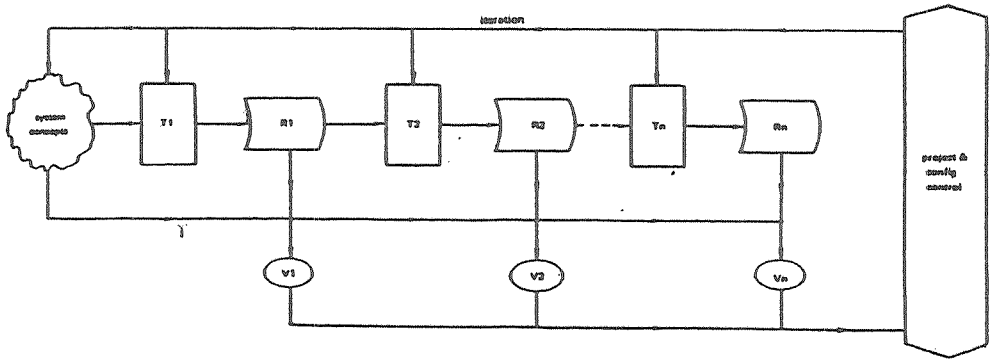
The European study started out with this view of the current land scape and tried to do a constructive experiment with a few methods which could be combined in a coherent way. In order to show the feasibility of a coherent integrated programming support environment, the choice of methods was guided by a couple of principles which had been formulated after literature studies, on the basis of our own experience, and on the basis of a large number of interviews with experts in the field, mainly in the USA.

The first of these principles is that complete formal treatments have only a limited feasibility for several reasons : a complete formal treatment of non-trivial problems is very difficult and hardly economical ; very often people favour operational descriptions ; different formalisms are needed to arrive at the most adequate formalization of specific problem domains. However, the knowledge of formal methods is beneficial for software development as the knowledge provides a profound understanding of the task.

The second principle is the need for flexibility, the need for having various representations, formalisms, methods, and tools which could also be used in parallel on the same project and even the same project phases. The third principle is that analysis play an important role : analysis should be available at every level of the software development life cycle ; it should allow for prototyping, and it may be based on processable annotation languages. The fourth principle is the need for customization and for reusing of software. In fact, software development is becoming so expensive that software should always be developed as a family of programs which can be adapted to various needs and reused as much as possible. The fifth principle is that production programming is very much different from experimental programming. Production programming, especially in an industrial context on demanding projects, needs life cycle management support and support for feasibility analysis, measurements, and quality assurance. On the basis of these principles, the study concentrated on the integration of methods, using existing work, and developing a partially original framework where details could be filled in later on.

2. THE SOFTWARE DEVELOPMENT PROCESS MODEL

Of course, I cannot help presenting a life cycle model here, but I think there are three good reasons for doing this. First, life cycle models are good for illustrational purposes ; however, they are just models and must not be over interpreted. Second, many people have one, and if they have one then they should rather have the good and correct one in order to get on the right track of thinking. And the third reason why I want to present another life cycle model is that it hints in the direction of various management aspects in the sense that it identifies canonical steps of production corresponding to the various activities on the various life cycle levels.



The model I'm showing you here is one which you havn't seen so far ; it's much more complicated than the waterfall model (Boe 76) ; it has more boxes and more arrows.

This life cycle model is based upon recognition of the need to employ multiple levels of abstractions in the development and maintenance of a software system. During the development process, several distinct representations are produced, one at each level of abstraction. The highest level on the left of our diagram must state the requirements for the system. These requirements should reflect the overall function of the complete system and the environment in which it is to operate. This first representation must give the complete picture without obscuring the description of what the system does with irrelevant details. At the lowest level to the right of the model, of course, there must be a representation which really can be executed. There is an immense gap between the two levels of requirements expression and of executable system so that it cannot be conveniently or reliably bridged in a single step. Therefore, several intermediate representations must be developed in order to master the complexity.

Development proceeds from the highest level representation to the lowest level one. It transforms each representation to produce its successor in the sequence. Transformation means taking a representation as the input and producing another representation as the output. It is important to know that this transformation might involve taking design decisions so that the

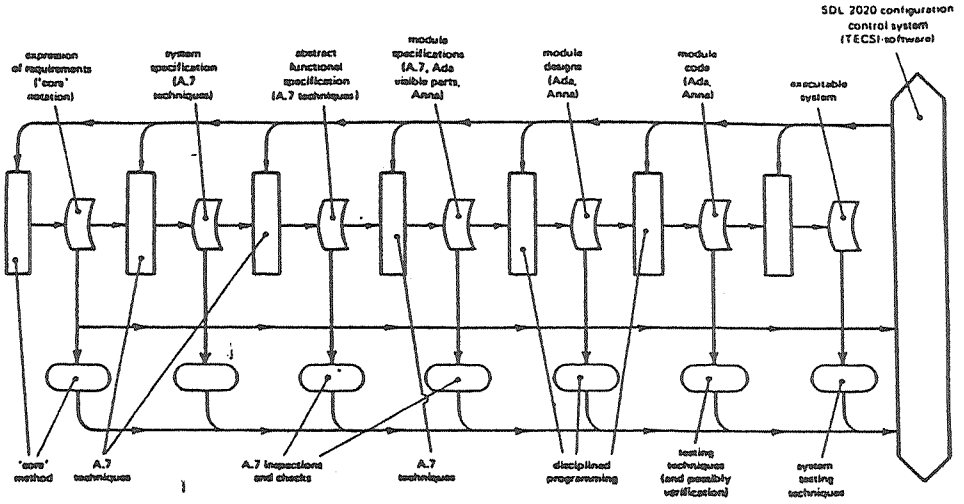
output representation may not be isomorphic to the input representation. The transformation may also be based on information taken from more than one preceding representation.

For an orderly development process it must be possible to verify each representation as it is produced. It has to be verified that a given representation is internally consistent and logically complete and that it is also consistent with the earlier representations in the sequence. The model shows that verification may involve references to any of the earlier representations in the sequence. Unfortunately, in reality the development of a complex system does not really procede in a straightforward fashion especially not if the development is going on for years. The development and verification of later representations will lead to the identification of errors and misguided design decisions which influenced the production of earlier representations. The requirements are also likely to change. Therefore, there is an important need for iteration around the representations which is reflected in the given model. On each iteration, some existing representation will be changed to correct the errors and to reflect the requirements changes, and the changed representation will again be verified. All these changes will be carried through to all later dependent representations and these will also be reverified.

Such an inevitable presence of iteration leads of course to non-trivial problems in retaining control of the overall process, and these are very often compounded by the fact that there is a need to develop several distinct versions of the given system. Thus there is a need for effective project and configuration control to which all development activities must be subjected. The project management functions are illustrated by the large box at the right end of the model. Project management is responsible for planning and controlling the development and maintenance processes which will lead to the production and modification of the representations. Of course this model as all the others does not show all the details of the development and management processes. But it can be used to explore the major characteristics of methods used during the life cycle. For it can be used to illustrate the level and purpose of the various representations that are employed, the form and notation used for each representation, the means of performing the transformation from each representation to the next, the way in which each representation is validated, the means by which iteration and change propagation are controlled, and the overall approach to configuration control.

3. DEVELOPMENT METHODS

So far the life cycle model has been left uninterpreted as to the nature of the system representations. The study gives a specific interpretation in connexion with the choice of existing methods which were investigated as to their suitability as candidates in a coherent environment. The interpretation chosen by the study is the following.



The first system representation at the left end of the life cycle model is the requirements expression. It is a description of the environment in which the system to be developed will work plus an outline of the intended function of the system. This representation is generated by extracting from the customer his requirements for the system. The verification involves checking by the customer that the representation accurately describes his needs. Consequently the requirement expression should be couched in a notation which can be understood by the customer.

The next representation of the system is the system specification. It is in essence a blackbox specification of the system behaviour. It must specify the input to and output from the system and specify the operations which the system performs in deriving the outputs from the inputs. This representation should be formal as it is intended for use by the system designer, not by the customer. The transformation from the requirements expression to the system specification consists of reexpressing part of the former and adding extra information obtained as necessary from further contact with the customer. The verification shows that the system specification is consistent with the relevant part of the requirements expression.

The next representation is the abstract functional specification, in fact the highest level of system design. It specifies the functions implemented by the system, the data flows within the system, and the data flows across the system boundary. This specification shows internal system structure. It can be considered as a set of interconnected black boxes rather than one black box. It still expresses a language independent design. The transformation from the system specification to the abstract functional specification is the skilled task of design. And the verification has to show that the abstract functional specification correctly implements the

system specification.

The next transformation step introduces specific programming language dependency. It is the transformation of the abstract functional specification to a module specification which is the language oriented design specification. It states the decomposition of the system into modules and the functions performed by each module. The transformation is thus a continuation of the design process refining the abstract functional specification and couching the design in terms of a specific high level programming language.

The next transformation step is going to specify the internal structure of the modules in the activity called module design. Again this representation is of course in a programming language oriented notation. From the module design the module code is produced which comprises the source text for the modules and additional annotations clarifying the semantics of the modules. The module code is transformed in the classical way by compilation, linking and loading into an executable system. The executable system may also include symbol tables for debugging, test harnesses etc. The verification of this last transformation is usually done by means of testing but may also occasionally be based on formal verification of the program properties. An interpretation of the large project management box at the end of the life cycle will be given in my next talk on the important role of software configuration management in a programming support environment.

The methods investigated by the study were the following : for the requirements expression, CORE of Systems Designers Ltd. was used (SyT 83) ; for the system specification and abstract functional specification, the "A7 methods" developed by Dave Parnas and his colleagues for the reimplementation of the A7 aircraft software were used (HKL 80) ; in the area of module specification, module design and coding the methods were those of structured programming but using the notations of Ada and ANNA (KBL 80). As the effort for the study was very limited verification methods could not be studied in detail.

The choice of the methods and notations mentioned was quite subjective but was based on the possibility of combining these methods in a coherent way. Local alternatives of these methods were identified but not examined closely. These alternatives are for instance : on the requirements expression level, the SADT method (Ros 77) ; on the system specification level, the Software Requirements Engineering Methodology (SREM) (Alf 77) ; on the level of the abstract functional specification, the Hierarchical Development Methodology (Sil 81) and/or Jackson's Structured Systems Analysis (see this workshop) ; and in the area of module specification, design, and coding, methods supported by the Program Development System (Harvard University) (Che 81) or the CIP project (Technical University of Munich) (see this workshop).

In order to provide more basis for discussion we shall briefly describe the methods used and investigated.

The CORE method for establishing requirements expression is expected to be used by an analyst who is trying to establish the requirements expression by illiciting information from a set of user representatives. CORE provides guidelines for establishing the relationships between the viewpoints of the system held by these representatives, known as a viewpoint hierarchy. The CORE method consists of establishing the viewpoint, then interviewing the user representatives or reading documents which describe their view of the system. The representatives are interviewed in an order based on the viewpoint hierarchy which enables the information, which they give, readily to be checked for consistency with the other views. A set of analyses are performed to ensure that the views are consistent and that they are analytically complete. An important aspect of the CORE analyses is that they make it clear when the requirements expression has been completed and the analysis should stop. The notation consists of diagrams supported by prose. There are diagrams showing data relationships, and a further set of diagrams which show functions, the functional dependencies, and the data flow between the functions. The notation is hierarchic in character and deliberately allows some ambiguity of expression, particularly concerning the characteristics of the data flows between functions. This notation is well-defined but semi-formal. We believe that formality of notation is generally desirable but that on the other hand the fully formal method is inappropriate to requirements expression. The CORE verification process is not formal. It consists of checking and acceptance by the customer authority. CORE is amenable to tool support. As the terminology of requirements expression is used in different ways it should once more be pointed out that this requirements expression is the system representation on the very highest level. It identifies the requirements and identifies the data and the functions of the system just sufficiently precisely enough to be able to document the understanding the customer authority has of the system to be developed.

The A7 methods can be used to provide a clear system specification which contains a very large part if not all of the information necessary to allow the system to be designed and built. The method uses a notation which is concerned with data items, functions, and the events which cause the functions to be applied to the data items. The notation allows for one level of functions rather than a hierarchy of functions. The data items are described in three distinct classes. Real word available information, i.e. data items which are available in the systems environment ; logical output data items, i.e. data items output by the system to the environment ; and auxiliary data items, i.e. data items retained by the system.

These data items are all described in a syntatic notation which define their logical form, and which may be used to specify the values which the data items can take. The functions are classed as being either demand or periodic with the obvious interpretation. The transfer characteristics of the functions are defined either by tables similar to truth tables, or by pseudo code. The mapping between the inputs and the outputs and the peripheral devices attached to the system are stated in the system specification. Finally, some auxiliary information which may be useful in the implementation is recorded ; for example, information on expected changes may help the guidance of the design on information hiding principles. The representation thus combines formal notation with prose. The verification within the system specification representation is formal

being primarily concerned with showing the completeness and correctness of the tables. The techniques are amenable to tool support. The A7 techniques give a systematic method for developing a system specification which is structured in such a way that the different parts of the specification are highly orthogonal. This means that the inevitable changes in specification can be accommodated with comparative ease. Further, the pragmatic formality of the techniques, i.e. the careful blend of formal and informal notations, makes them easy to learn and apply but nonetheless yields highly precise specfications. What we have called system specification here is very often also called requirements specification.

The A7 methods are also applied to produce the abstract functional specification from the system specification. This transformation step is not neccessary straightforward ; it depends heavily on the skill of the designer. Design guidelines may be information hiding and design for ease of contraction and extension. The main problem is that the transfer characteristics of composite functions have to be deduced from the control flow and the transfer characteristics of the functions being composed. This transformation step also has to take into account performance aspects especially if the system is to be an embedded system consisting of several processors.

The transformation from the abstract functional specification to the module specification was seen as the step to express the functions on the abstract functional specification level in Ada using the Ada specification constructs and annotating these Ada constructs with ANNA statements. ANNA is a notation for annotating Ada programs with packages. It is based on the predicate calculus but includes extensions to deal with concepts, such as array update, which are foreign to the predicate calculus. An ANNA specification for a package states invariants for data objects and for the subprograms visible in the package.

ANNA does allow one to perform the basic task of module specification, that of specifying the semantics of the subprograms within each module, with reasonable accuracy. ANNA is not entirely satisfactory for module specification as it stands because of its inability to deal with tasks. Even if this short-coming were resolved, the use of ANNA still can be considered dubious primarily because of the low level of abstraction, i.e. the high level of detail, at which an annotation in ANNA has to be done. The biggest deficiency in the ANNA version which we used (that of 1980) is the lack of a mechanism for handling abstraction without explicitly introducing it by auxiliary Ada text. ANNA will be very effective if one wants to perform verification. ANNA provides enough formality, and adequate tool support can be developed for ANNA which would be very beneficial.

Ada and ANNA were also used as notations to detail the design in the next transformation step leading from the module specification to the module design, and, of course, Ada was used to code the system. On the module design level, ANNA seems to be much more suitable because the level of abstraction is lower than on the module specification level.

Presenting the methods, we have mentioned a number of transformation steps. Each of these steps has to be checked in the sense that one tries to show that the low level representation is a plausible extension of the higher level representation. The lower level representation contains more information than the higher level one. The recording of mappings of elements of the higher level to elements on a lower level and the recording of decisions enriching the lower level represenation are both necessary and useful. Also the plausibility proof of the transformation should be recorded during the construction. If one can show that a transformation leads to a satisfactory expansion of the representation than this does not yet mean that one shows that one does this expansion in a quality way. To show that the design is of good quality, criteria and methods have to be developed.

During the study an experimental development was performed using the methods and notations presented. Furthermore, the conclusions from this experiment were that tool support for methods is very much critical especially if there is a large number of representations which very often restate properties again and again. The experiment has shown that there are many benefits from integration. Project visibility and control are improved, the system complexity is mastered more easily, confidence in correctness is increased, and making changes becomes easier. Of course, it should be stressed at this point once more that the study was quite subjective in its choices and also very limited in its effort. Fortunately, the investigation has also shown that coherency leads to the possibility of providing much more extensive tool support than could be provided without coherency of methods. It is worth while to make a few remarks about the kinds of tools one should develop for a programming support environment. In our opinion these tools can be classified into basic, aggregated, and integrated support tools.

Basic support tools are tools which are primarily concerned with syntactic aspects of the notations. Such tools may be syntax-directed editors, syntax checkers, and report generators. These tools could be produced from generic tools which are generators of editors or generators of syntactic analysers.

Aggregated tools are tools which have each knowledge of the semantics of one representation. They can check for the consistency and completeness of representations, enforce standards, and analyse representations.

Integrated support tools are tools which know about the semantics of several representations and make use of the relationships between representations. Such tools are for instance drivers of the transformation process from one representation to the other, program provers, rapid implementation tools, and tools for symbolic execution, modeling, simulation, and performance analysis.

Most of the support tools could be developed using existing technology, and most of them would make use of basic tools which are accessing the information about the representations in a software engineering data base. The various tools are discussed in more detail in the study report

(SyT 83) where a progressive development plan for a programming support environment is discussed.

4. CONCLUSIONS

In this presentation, we have hardly been able to provide enough substance to make you accept our claim that an integrated programming support environment of the kind described is feasible, and that there is much benefit from coherence. We believe that rapid progress is possible if the opportunity given today, especially with the advent of Ada, is seized to develop coherent environments, and if this chance is seized using good existing methods.

It is very important to provide industry with rapid improvements. But industry does need practical solutions, not the most sophisticated solutions. We are furthermore convinced that already clerical support in a flexible way would lead to substantial improvements, and, therefore, have a very high pay-off ratio. By clerical support we mean simple support, for instance, for creating and maintaining documentation about representations on the computer. Today, industry still works mainly on paper, controlling thousands of pages of paper in their evolution with a lot of effort and difficulty.

But even such clerical support should be provided in a flexible way which is independent of specific methods or tools as far as possible since each organization uses different methods, different overall methodologies, and different hardware with different tools. This is the situation today, which we cannot change completely from one day to the other. The transition has to be smooth, and can only take place over quite a long period.

Developing clerical support in a flexible way means avoiding turnkey solutions which would stifle progress. The environment tools today should mainly be generic components and generators of specific components which could be adapted to evolving needs, notations and methods. There should be a whole range of such generic components so that a customization of support environments becomes possible. In fact, this means, and this may a good conclusion, that the development of support environments has, in the first place, to be guided by the principles for engineering large software systems.

5. REFERENCES

(Alf 77) Alford, M.W., A requirements engineering methodology for real-time processing requirements, IEEE Trans. on Software Enginering. SE3 (1977) 60-69.

(Boe 76) Boehm, B.W., Software Engineering, IEEE Transactions on Computers, C-25, n° 12, 1976.

(Che 81) Cheatham, T.E., An Overview of the Harvard Program Development System, In : (Hün 81) 253-266.

(HKL 80) Heninger, K.L., Kallander, J.W., Shore, J.E., and Parnas, D.L., Software requirements for the A-7E aircraft, Naval Research Laboratory, Washington D.C. (1980).

(Hün 81) Hünke, H., (Ed.), Software Engineering Environments, North Holland, Amsterdam, 1981.

(KBL 80) Krieg-Brückner, B., Luckham, D.C., ANNA : towards a language for annotating Ada programs, ACM Sigplan Notices 15, 11 (1980) 128-138.

(McR 84) McDermid, J., Ripken, K., Life Cycle Support in the Ada Environment, Ada Companion Series, Vol. I, Cambridge University Press, Cambridge, 1984.

(Met 82) Methodman, Software Development Methodologies and Ada, US Department of Defence, 1982

(Ros 77) Ross, D.T., Structured Analysis (SA) : a langage for communicating ideas, IEEE Trans. on Software Engineering. SE-3 (1977) 16-34.

(Sil 81) Silverberg, B.A., An overview of the SRI Hierarchical Development Methodology, in (Was 81), 211-227.

(Sto 80) Stoneman, Requirements for Ada Programming Support Environments, US. Dept. of Defence, Feb. 1980.

(SyT 83) Systems Designers Limited, Tecsi-Software, Life-Cycle Support in the Ada Environment, Final Report of a Study for the CEC, Fleet, Paris, March 1983.

(Was 81) Wasserman, A.I., (Ed.), Tutorial : Software Development Environments, IEEE, New York 1981.

TO BE PUBLISHED THIS NOVEMBER

Life Cycle Support in the Ada Environment

J.A. McDERMID
System Designers Ltd., Fleet
and K.RIPKEN
TECSI Software, Paris
on behalf of the Commission of the European Communities

Integrated Ada Programming Support Environments (APSE) based on coherent methods covering all stages of the software life-cycle, and supporting programmers, software engineers and their managers with harmoniously cooperating toolsets were envisaged as early as 1980. However, to date only minimal versions (MAPSE) have been undertaken, concentrating on support for compilation and execution.

In 1982, SDL and TECSI, with the support of the Commission of the European Communities, performed a ground-breaking study that has taken matters one step further: they performed an initial top-down design of a more-than-minimal APSE stressing full life-cycle coverage, coherence of methods and of tools, and integration of management and production.

The final report of the study has been adopted as the opening volume of the new Ada Companion Series. Broad in scope, and convincing in its detail, the report presents a detailed life-cycle model, discusses a management philosophy compatible with that model, and gives an experimental assessment of individual methods (notably CORE and A-7). It concludes that a coherent APSE is possible within state-of-the-art technology, and its development should be undertaken soon to reap the benefits of its use in production of large-scale software systems.

Contents

Ada Companion Series

228 x 152 mm. about 250 pp.
0 521 26042 6 Hard covers about £12.50 net

CAMBRIDGE UNIVERSITY PRESS

ANNOUNCING AN INVALUABLE NEW SERIES OF COMPUTER SCIENCE BOOKS

The Ada Companion Series

There are currently no better candidates for a co-ordinated, low-risk, and synergetic approach to software development than the Ada programming language. Integrated into a support environment, Ada promises to give a solid standards-orientated foundation for the higher professionalisation of software engineering.

This definitive new series aims to be the guide to the Ada software industry for managers, implementors, software producers and users. It will deal with all aspects of the emerging industry: adopting an Ada strategy, conversion issues, style and portability issues, and management. To assist the organised development of an Ada-orientated software components industry, equal emphasis will be placed on all phases of life cycle support.

Volumes on style and portability, language conversion, and Ada and multiprocessor systems are in preparation. The first volume to appear provides an appropriate life cycle model.

Please order from your bookseller or, in case of difficulty, from CAMBRIDGE UNIVERSITY PRESS, The Edinburgh Building, Shaftesbury Road, Cambridge CB2 2RU, U.K.; or CAMBRIDGE UNIVERSITY PRESS, American Branch, 32 East 57th Street, New York, NY 10022, U.S.A.

Editorial enquiries should be addressed to Dr. Ernest Kirkwood at the Cambridge office, or to the Secretary of the Editorial Board

- WIDE RANGE OF AVAILABLE "SUPPORT"

  BUT EXISTING SE^2s :

  - EACH ALONE
    - INCOMPLETE
    - CLOSED
    - POOR TEAM SUPPORT
  - IN COMBINATION INCOMPATIBLE
  - OFTEN INCOMPATIBLE WITH ADA

DIFFICULTIES COMPOUNDED BY DIFFERENT

- TERMINOLOGY
- UNDERSTANDING OF CONCEPTS AND OBJECTIVES
- PRIORITIES

PURPOSE OF STUDY

- PRODUCE A STAWMAN-LEVEL DOCUMENT

  FOR APSE DESIGN AND DEVELOPMENT

  - PRELIMINARY
  - BASIS FOR DISCUSSION
  - OPTIONS
  - INDICATION OF AREAS OF FURTHER STUDY

    OR ENHANCEMENT

    OR ADAPTATION TO ADA

- EMPHASIS ON
  - FULL LIFE CYCLE
  - PROJECT TEAM
  - GENERAL INTEGRATION
  - FLEXIBLE COMBINATION OF GOOD EXISTING METHODS
  - PRACTICAL TECHNIQUES
  - ADAPTATION WHERE NECESSARY
- ACHIEVEMENTS
  - FEASIBILITY, BENEFITS, PROBLEMS
  - BASE/CLASSIFICATION OF TERMINOLOGY, CONCEPTS, OBJECTIVES
  - IDENTIFICATION OF FURTHER WORK

REQUIREMENTS EXPRESSION

SYSTEMS SPECIFICATION

ABSTRACT FUNCTIONAL SPECIFICATION

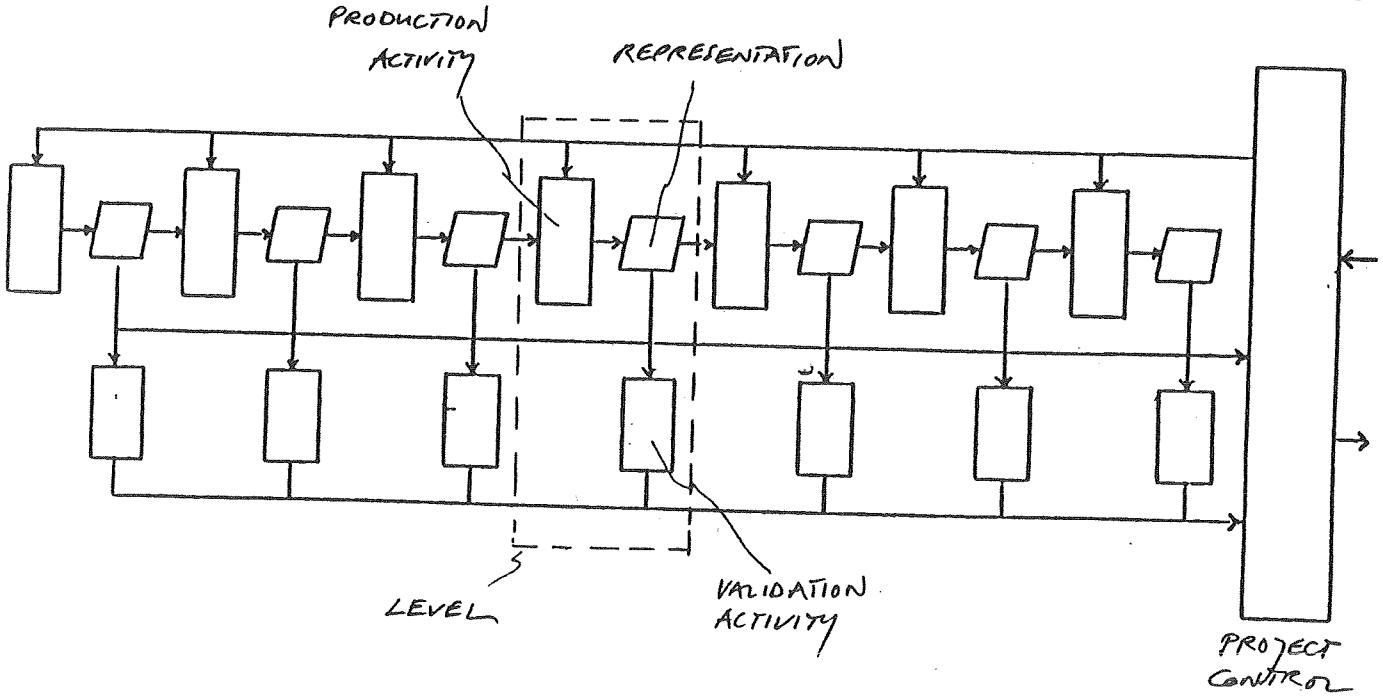
MODULE SPECIFICATION

MODULE DESIGN

MODULE CODE

EXECUTABLE SYSTEM

DATABASE



MAIN ROAD INVESTIGATED

REQUIREMENTS EXPRESSION

SYSTEMS SPECIFICATION

ABSTRACT FUNCTIONAL SPECIFICATION

MODULE SPECIFICATION

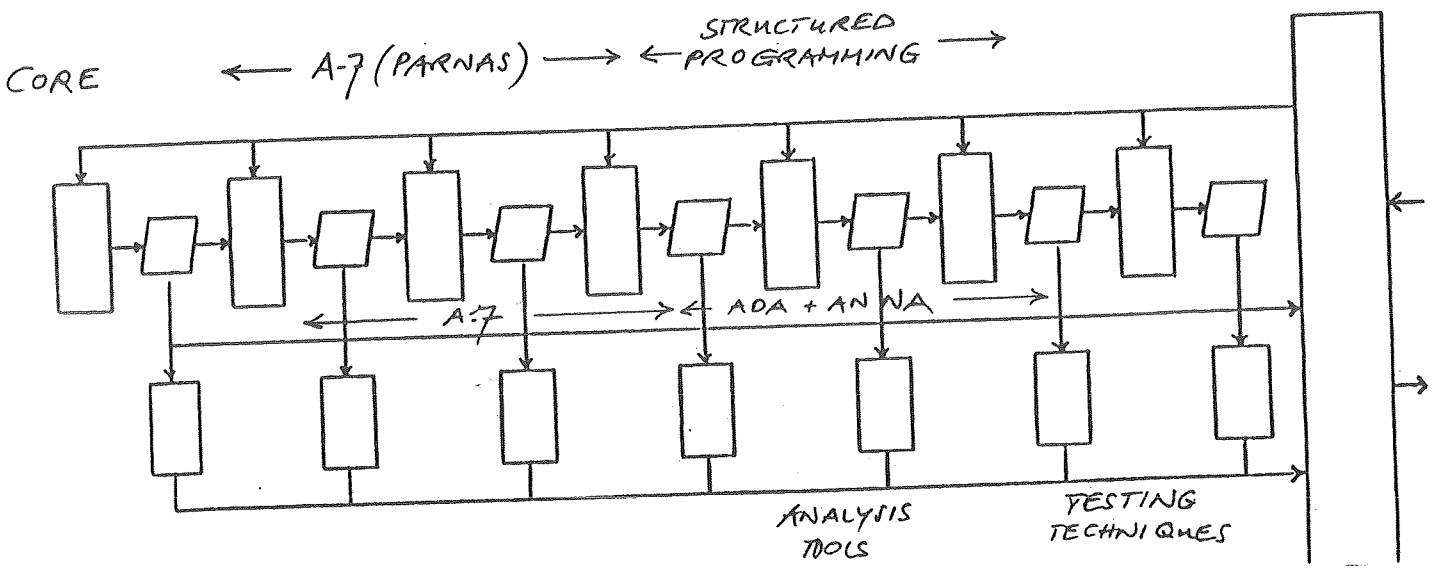
MODULE DESIGN

MODULE CODE

EXECUTABLE SYSTEM

DATABASE

SDL 2020



LOCAL ALTERNATIVES

REQUIREMENTS EXPRESSION

SYSTEMS SPECIFICATION

ABSTRACT FUNCTIONAL SPECIFICATION

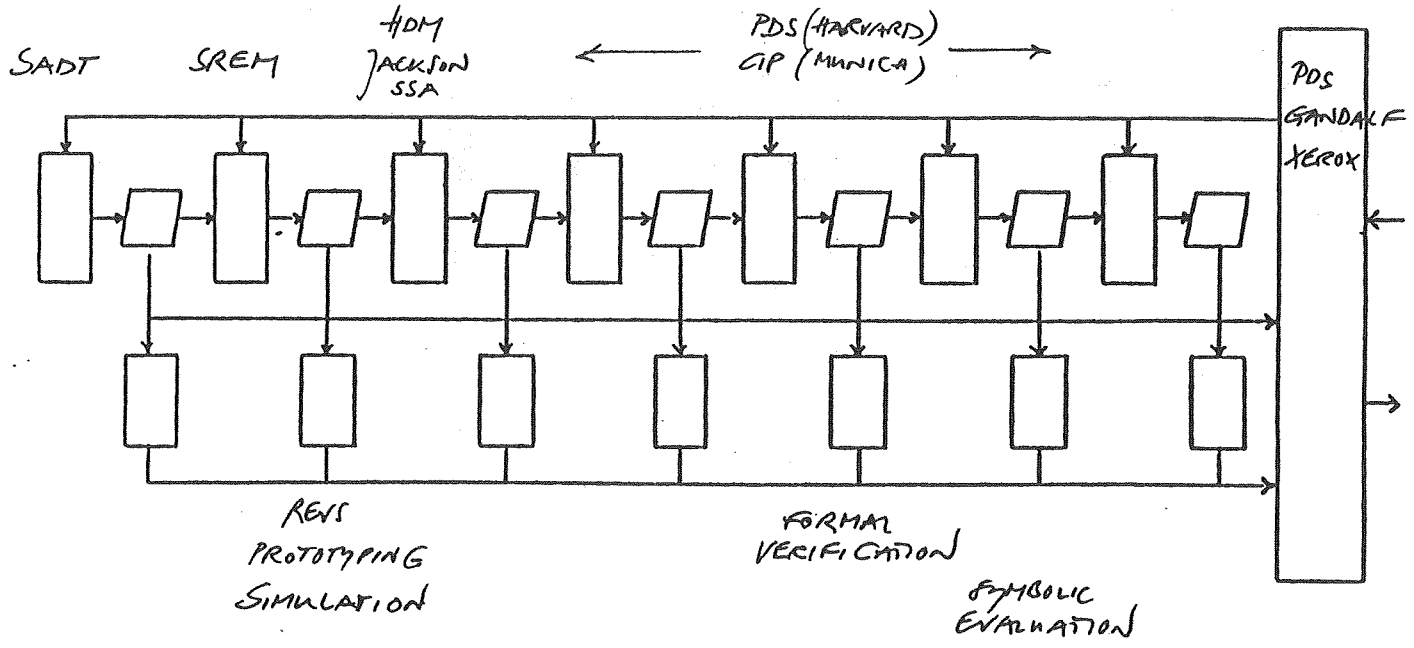
MODULE SPECIFICATION

MODULE DESIGN

MODULE CODE

EXECUTABLE SYSTEM

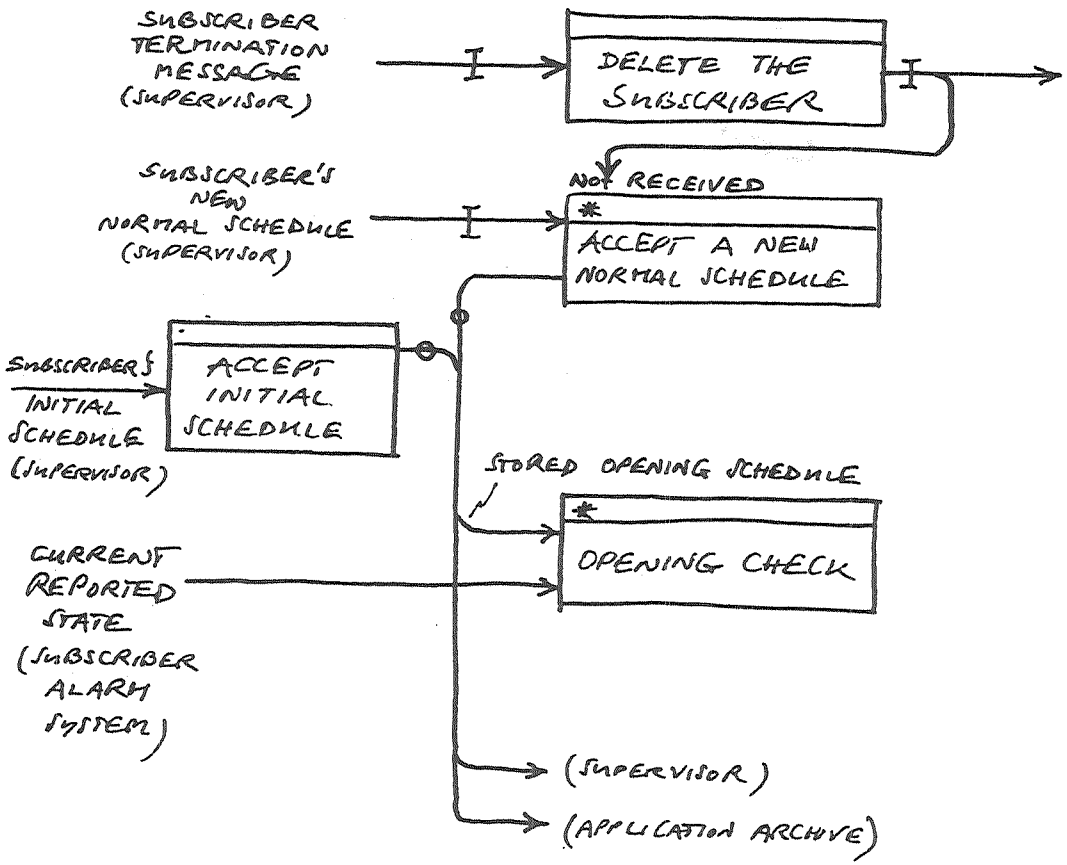
DATABASE



REQUIREMENTS EXPRESSION

CORE

- to elicit information from user representatives
- viewpoint hierarchy
- reliability assessment
- termination
- semi-formal diagrams & prose
- informal verification
- easy change
- tool support easily possible



SYSTEM OPERATION (INCOMPLETE)
(BURGLAR ALARM SYSTEM)

OPERATIONAL FLIGHT PROGRAM (OFP) OF NAVY-A7 AIRCRAFT

- TYPICAL FOR DoD
- CHARACTERISTICS :
  - BARELY MEETS TIME & SPACE LIMITATIONS
  - NOT FULLY UNDERSTOOD BY MAINTENANCE PERSONNEL
  - POORLY DOCUMENTED
  - DIFFICULT TO CHANGE

⇒ JOINT PROJECT OF NAVAL RESEARCH LAB. AND NAVAL WEAPONS CENTER TO REDESIGN AND REBUILD THE A-7 OFP

- DEMONSTRATE USEFULNESS OF "GOOD" SOFTWARE ENGINEERING TECHNIQUES
- PROVIDE A FULLY WORKED OUT EXAMPLE AS A MODEL

SYSTEM SPECIFICATION

A-7

- one level of functions
- data items (RWAI, LODI, AOI) logically defined
- transfer characteristics in tables
- formal verification
- amenable to tool support

Also used in

ABSTRACT FUNCTIONAL SPEC.

A-7 SPECIFICATION TECHNIQUE

/ input /

// output //

/// auxiliary data items ///

$ value $

! term !

* mode * abbreviation for a conjunction of conditions

event:

@T(condition)

@F(condition)

@T(condition$_1$) when (condition$_2$)

relational operators

standard functions

USE of TABLES

- promote Completeness
- achieve Concisenes

SELECTOR TABLE

MODE \ LOD1	//A//	//B//
M	V	U
N	X	W

↯ no statement being made

CONDITION TABLE

(defines the time intervals when values hold)

MODE	CONDITIONS		
M	A	B	C
N	B	C	X
DEFINITION OF !E!	U	V	W

DEFINITION exists appropriate for mode *N*

EVENT TABLE

MODE	EVENTS	
M	X	@T(Condition$_1$)
N *O*	@F(IN MODE)	@T(IN MODE) when (Condition$_2$)
ACTION:	TURN LIGHT ON	TURN LIGHT OFF

LOGICAL INPUT DATA ITEM

INPUT ITEM : SUBSCRIBERS NEW NORMAL SCHEDULE DETAILS

ACRONYM : /NEW_SCHED_DETAILS/

SOURCE : SUPERVISOR

DESCRIPTION : Updates the opening schedule for an existing subscriber

DATA REPRESENTATION :

/NEW_SCHED_DETAILS/ ::= <SUBSCRIBER_IDENTIFIER>
<OPENINGS_LIST>
NEW_SCHED_KEY

<OPENINGS_LIST> ::= ...

LOGICAL OUTPUT DATA ITEM

OUTPUT ITEM : CURRENT SCHEDULE

ACRONYM : // CURR_SCHED //

DESTINATIONS : SUPERVISOR, APPLICATION ARCHIVE

DESCRIPTION : ...

DATA REPRESENTATION :

// CURR_SCHED // ::= <SUBSCRIBER_IDENTIFIER>
<OPENINGS_LIST>
IS_CURRENT_SCHEDULE
DATE_TIME

FUNCTION

DEMAND FUNCTION NAME : UPDATE SCHEDULE

INPUT ITEMS : /NEW_SCHED_DETAILS/
/// SUBS_ACTIVE ///

OUTPUT ITEMS : // CURR_SCHED //

INITIATION CONDITION : @T(PRESENT(/NEW_SCHED_DETAILS/))

TERMINATION CONDITION : end of action

FUNCTION :

CONDITION \ LODI	// CURR_SCHED //
/// SUBS_ACTIVE /// (/SUBS_ID/)	(/SUBS_ID/, /OPENINGS_LIST, DATE_TIME)
not ...	X

ABSTRACT FUNCTIONAL SPECIFICATION

⋮

DEMAND FUNCTION : INITIATE_OR_UPDATE
INPUTS : /SCHED_DETAILS/, ...
OUTPUTS : // CURRENT_SCHEDS //, ...
LOCALS : /// SCHED_OK /// initial FALSE,
...

FUNCTION: INVOKES THE DEMAND FUNCTIONS

CHECK_SUBS
CHECK_SCHED
UPDATE_SUBS_ACTS
UPDATE_SCHEDULE
PRINT_AND_DISPLAY

} BASIC FUNCTIONS

⋮

UPDATE_SCHEDULE
INITIATION : @T (/// SCHED_OK ///)
FORMAL/ACTUAL PARAMETER MAPPING :

FORMAL	/SCHED/	/CURRENT_SCHEDS/	// CURRENT_...
ACTUAL	/SCHED/	/// CURRENT_SCHEDS ///	/// CURRENT_...

MODULE SPECIFICATION

ANNA

- based on predicate Calculus with additional Concepts
- package annotations state invariants of data and visible operations
- level of abstraction too low
  - not easy to modify
  - auxiliary Ada text to be introduced
- with adequate tool support beneficial

Also used in

MODULE DESIGN

```
with SUBSCRIBER_SCHEDULES;
   :
package INITIATION_AND_UPDATE is
   procedure INIT_OR_UPD (R: in REPORT,
                          V,L : out REPORT);
--| where out L = out V
--| and
--|    (( not (in R.SUBS_ID > 0
--|             and not R.SUBS_ID > MAX_SUBS)
--|     and out V = ( SCHED_PKG,
--|                   in R.SUBS_ID,
--|                   in DATE_TIME,
--|                   in R.SCHED
--|                   INVALID_SUBS_ID
--|                 )
--|     )
--|     or
--|        ( in R.SUBS_ID > 0
--|          and not in R.SUBS_ID > MAX_SUBS
--|          and ( ( in SUBS_ACTIVE (in R.SUBS_ID)
--|                   and in R.TYPE = INIT_SCHED
--|                   and out V = ( :
--|                                 SUBS_ALREADY_
--|                                   ACTIVE
--|                               )
--|              )
--|              or ...
```

```
package STACK is                          VIRTUAL Ada TEXT
                                                  N
   --: function LENGTH return NATURAL;
   procedure PUSH (X: ITEM);
   --| in STACK.LENGTH < SIZE;                    ANNOTATION
   :
end STACK;
```

```
package body STACK is
   type TABLE is array ( ... ) of ITEM;
   ...
   INDEX : NATURAL range 0.. SIZE := 0;

   --: function LENGTH return NATURAL;
   --| return INDEX;

   :
end STACK;
```

CONCLUSIONS

- BENEFITS : VISIBILITY, CONSISTENCY
- TOOL SUPPORT IS CRITICAL.
- APSE PROVIDES OPPORTUNITY.

INDUSTRY NEEDS RAPID IMPROVEMENTS
IN THE RIGHT DIRECTIONS :

- CLERICAL SUPPORT → HIGH PAY-OFF
- TECHNOLOGY EXISTS

  ! COHERENCY ⟶ PREMATURE FREEZING
- APPLY SE-PRINCIPLES :
  - METHOD-INDEPENDENT COMPONENTS
  - NO TURN-KEY SOLUTIONS
  - GENERIC COMPONENTS FOR CUSTOMIZATION & UPGRADING

Un Linguaggio algebrico di specifica e applicazioni

EGIDIO ASTESIANO

ISTITUTO DI MATEMATICA
GENOVA

ASL (An Algebraic Specification Language)
M. Wirsing 1983
universal algebraic specification language

SMoLCS (Structured Monitored Linear Concurrent Systems)
methodology for Cnet project
E. Astesiano, G. Reggio 1982 →
(+ E. Zucca, F. Mazzanti, U. Montanari)
operational modular hierarchical spec. of concurrency

ASL-SMoLCS (Making SMoLCS algebraic)
parameterized algebraic specification of concurrency
E. Astesiano, G.F. Mascari, G. Reggio, M. Wirsing 1984

ASL distinctive features

Kernel language for building high level spec. lang. (CLEAR, ...)
higher order - well-typed
based on (denotational) semantic operators

- Semantics of a spec is a set of algebras
- ability of spec. infinite sorts and signatures (all rec. en.)
- algebras are generalized heterogeneous algebras

- parameterization is achieved by λ-abstraction
  if a spec. T has T' as a parameter:
  specfunct $T = \lambda$ spec T', ...

more generally: ASL has all features of an applicative universal language for defining all computable transf. of spec.

- language for defining behavioural specification
  (behaviour w.r.t. a set of terms - not only initial or final spec.)

- simple notion of implementation
  roughly: T' is an impl. of T iff models of $T' \subseteq$ models of T
  - that notion extends to parameterized spec (with horizontal and vertical comp.)
  - from that notion almost all developed notions of impl. can be derived (models of $\mathcal{F}(T') \subseteq$ models of $\mathcal{G}(T)$)

ASL Modes (types of objects)

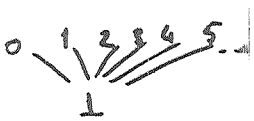
<object mode> ::= bool | nat | ...

<ground basic mode> ::= sort | opname | oper | term | formula | var | varname | <object mode>

<mode> ::= <g. b. mode> | sig | spec | sigmorph | funct(<mode>$^+$)<mode>
<mode>s | <mode> × <mode> | <mode> + <mode> | seq<mode>

Semantic domains of modes DOM[m]

- domains are cpos (for handling recursion)
- domains of ground basic modes are flat cpos



- domains of set modes are Plotkin powerdomains

$\mathrm{DOM}[ms] = \mathbb{P}_f(\mathrm{DOM}[m], \subseteq)$ (for allowing infinite sorts and signatures)

$S \in \mathbb{P}_f(D) \Rightarrow S = (S_1, S_2)$ S_1 positive information, S_2 negative information

$S_1, S_2 \in \mathbb{P}_{BM}(D)$

DOM[spec] = SEM
SEM = $(\{\langle \Sigma, C\rangle \mid \Sigma \in SIG \wedge C \subseteq Alg(\Sigma)\}, \leq)$
$\langle \Sigma, C\rangle \leq \langle \Sigma', C'\rangle$ iff $\Sigma \subseteq \Sigma' \wedge \forall A \in C' : A|_\Sigma \in C$

Operations on modes (syntax and semantics)
selectors, constructors, tests, ...

ASL Basic Operators (Specification expressions)

Syntax

<basic spec> ::= signature <sigexp> axioms <formexp>

< sum > ::= <spec exp> + <spec exp>

<reachable> ::= reachable <spec exp> on <sort exp> with <opns exp>

<derive> ::= derive from <spec exp> by <sigmorph exp>

<observe> ::= observe <spec exp> wrt <terms exp>

Semantics

$[\![\text{signature } \Sigma \text{ axioms } E]\!] = bspec(M[\![\Sigma]\!], M[\![E]\!])$

$[\![T + T']\!] = M[\![T]\!] + M[\![T']\!]$

$[\![\text{reachable } T \text{ on } S \text{ with } F]\!] = reach(M[\![T]\!], M[\![S]\!], M[\![F]\!])$

$[\![\text{derive from } T \text{ by } \sigma]\!] = derive(M[\![T]\!], M[\![\sigma]\!])$

$[\![\text{observe } T \text{ wrt } W]\!] = M[\![T]\!]|^{M[\![W]\!]}$

—

not the same as in CLEAR (but $+_{CLEAR}$ can be defined)

reachable does not imply initiality (as for ordinary data of CLEAR)

observe : $T|^W$ = {all Σ-algebras W-equivalent to some model of T}

$A \equiv_W A'$ iff $\left[\begin{array}{l} A \models t = t'[v] \Leftrightarrow A' \models t = t'[v'] \\ A \models D(t)[v] \Leftrightarrow A' \models D(t)[v'] \end{array}\right.$ for some surjective valuations v, v'

many different observation equivalences can be derived by specializing W

derived operators ex.: enrich T by sorts S opns F axioms E

Specifying and modelling concurrency

— two approaches to specification
(see Lamport's IFIP '83 talk for a discussion)

- axiomatic approach (temporal logic, ...)
  subclass of models in a predefined class

- constructive approach
  a model (CCS, CSP, ...)

axioms ⟶ derive and prove correct implementations (models)

model ⟶ abstract from impl. details (usually giving equivalences w.r.t some observ. conditions)

— for purely specif. purposes axiom. appr. is more suitable (especially for refinement/hierarchical methodologies)

for modelling and study properties of concurrency a constructive approach has given good results (see CCS)

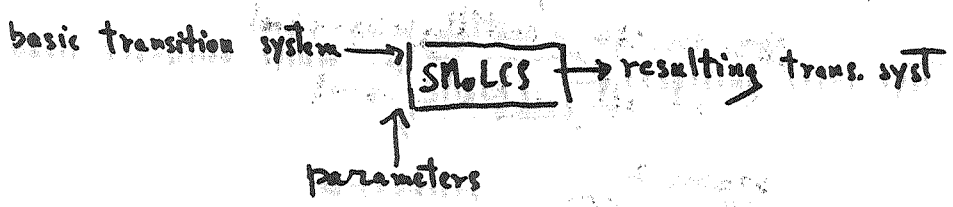
it is widely recognized (industry people, also C. Jones' various papers) that a combination of the two would be nice

perfect approach: unifying the two (as in "static" data types sometimes happens)

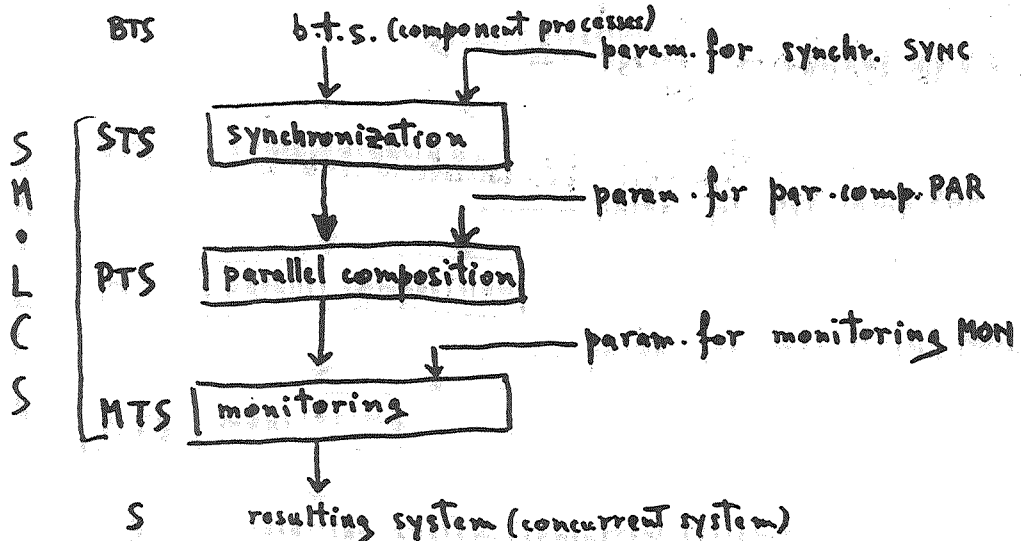
— our approach is (constructive, but) between ⟶ axiomatic / constructive style à la CCS, CSP

SMoLCS operational schema

- specification of a process = (finite presentation of a) flagged transition system (fts)
- specification of a concurrent system = (spec. of the) composition of component processes into an fts
  basic transition system
- schema = parameterized composition



- three-step composition



S = SMoLCS(BTS, SYNC, PAR, MON) =
= MTS(PTS(STS(BTS, SYNC), PAR,) MON)

Making SMoLCS algebraic

motivations
- precise formalization of the parameterization schema
- abstraction → for data structures (abstract data type style & motiv
  → for different observational semantics

tools
- adt specification of static structures (states, flags, —)
- transition systems as adt (adt spec. of dynamic struct.)
- parameterized schema as a parameterized adt (unique)
  SMoLCS system = instantiation on appropriate parameters
  (in ASL parameterized adt = specification function)
- semantics as specification of observable features
  (in ASL via the "observe" construct)

novelty
- parameterization schema
- three-step procedure

related w.
- Broy-Wirsing spec. of CSP
- Plotkin's SOS methodology
- Mosses' BASA for seq. languages

Transition systems

spec-funct TS ≡ λ spec STATE, FLAG

formulas E_T: E_T is "⟨cond⟩ $s \xrightarrow{t} s'$"

enrich Bool + STATE + FLAG by

ops $\_\Rightarrow\_ : \text{state} \times \{\text{flag}\}^* \times \text{state} \to \text{bool}$

axioms E_T

Hierarchical transition systems

specfunct HTS = λ spec BTS: IS-TS[BSTATE, BFLAG, E_T, →]

spec STATE, FLAG

formulas E_H: E_H is "⟨cond⟩ $\bigwedge_{j=1}^{m} bs_j \xrightarrow{bf_j} bs'_j \; \wedge \; s \xrightarrow{t} s'$"

BTS + TS(STATE, FLAG, E_H)

where

specfunct F = λ spec BTS: IS-TS[BSTATE, BFLAG, E_T, →].sup

abbreviates

specfunct F = λ spec BSTATE, BFLAG

formulas E_T

rename →

spec BTS: TS(BSTATE, BFLAG, E_T)[→/⇒].sup

Synchronization

specfunct STS ≡ λ spec BTS: IS-TS[STATE, FLAG, E_T, →]

spec INF, SFLAG

formulas E_{SST}, E_{SYNC}

HTS(BTS, SST, SYNC⟨⟨flag⟩⟩, E_S)

where

SST = PRODCOND(MSET(STATE), INF, E_{SST})

SYNC = enrich MSET(FLAG) + SFLAG + INF by

ops iss: ⟨flag⟩ × mset(flag) × inf → bool

sit: ⟨flag⟩ × inf → inf

axioms E_{SYNC}

E_S =

$\{ iss(\langle f_1, f_2 \cdots f_m \rangle, i) = true \;\; \bigwedge_{j=1}^{m} s_j \xrightarrow{f_j} s'_j \;\; > \;\; \langle s_1 | \cdots | s_m, i \rangle \xrightarrow{sf} \langle s'_1 | \cdots | s'_m, sit(\langle f_1 \cdots \rangle, i) \rangle \mid m > 0 \}$

Parallel composition

specfunct PTS ≡ λ spec BTS: IS-TS[PST: IS-PRODCOND[MSET(STATE, INF, E_{PST}], FLAG, E_{T_1}, →] ∧ (→ = ⇒)

formulas E_{PAR}

HTS(BTS, PST⟨prod⟩, PAR⟨flag⟩, E_P)

where

PAR = enrich FLAG + INF by

ops //: flag × flag → flag (comm., assoc.)

pit: flag × inf → inf

axioms E_{PAR}

E_P = { $D(f_1 // f_2) \bigwedge_{j=1}^{n} \langle ms_j, i\rangle \xrightarrow{t_j} \langle ms'_j, i'_j\rangle$ ⊃

$\langle ms_1 | ms_2, i\rangle \xrightarrow{t_1 // t_2} \langle ms'_1 | ms'_2, pit(f_1 // f_2, i)\rangle$ }

Monitoring

specfunct MTS ≡ λ spec BTS: IS-TS[STATE: IS-PRODCOND[MSET(BSTATE), INF, E_{ST}][list/prod], FLAG, E_{T_2}, →]

spec MINF, MFLAG

formulas E_{MST}, E_{MON}

HTS(BTS, MSTATE⟨prod⟩, MON⟨mflag⟩, E_M)

where MSTATE = PRODCOND(STATE, MINF, E_{MST})

MON = enrich BTS + MINF + MFLAG by

ops mon: flag × list × minf → mflag

mit: flag × minf → minf

axioms E_{MON}

E_M = { $mon(f_j, \langle \overline{ms}, i\rangle, mi) = mf \wedge \langle ms, i\rangle \xrightarrow{t} \langle ms', i'\rangle \wedge ms \leq \overline{ms}$ ⊃

$\langle\langle \overline{ms}, i\rangle, mi\rangle \xrightarrow{mt} \langle\langle (\overline{ms} - ms) | ms', i'\rangle, mit(f, mi)\rangle$ }

Parallel composition

specfunct PTS ≡ λ spec BTS: IS-TS[PST: IS-PRODCOND[MSET(STATE, INF, E_{PST}], FLAG, E_{T_1}, →] ∧ (→ = ⇒)

formulas E_{PAR}

HTS(BTS, PST⟨prod⟩, PAR⟨flag⟩, E_P)

where

PAR = enrich FLAG + INF by

ops //: flag × flag → flag (comm., assoc.)

pit: flag × inf → inf

axioms E_{PAR}

E_P = { $D(f_1 // f_2) \bigwedge_{j=1}^{n} \langle ms_j, i\rangle \xrightarrow{t_j} \langle ms'_j, i'_j\rangle$ ⊃

$\langle ms_1 | ms_2, i\rangle \xrightarrow{t_1 // t_2} \langle ms'_1 | ms'_2, pit(f_1 // f_2, i)\rangle$ }

Monitoring

specfunct MTS ≡ λ spec BTS: IS-TS[STATE: IS-PRODCOND[MSET(BSTATE), INF, E_{ST}][list/prod], FLAG, E_{T_2}, →]

spec MINF, MFLAG

formulas E_{MST}, E_{MON}

HTS(BTS, MSTATE⟨prod⟩, MON⟨mflag⟩, E_M)

where MSTATE = PRODCOND(STATE, MINF, E_{MST})

MON = enrich BTS + MINF + MFLAG by

ops mon: flag × list × minf → mflag

mit: flag × minf → minf

axioms E_{MON}

E_M = { $mon(f_j, \langle \overline{ms}, i\rangle, mi) = mf \wedge \langle ms, i\rangle \xrightarrow{t} \langle ms', i'\rangle \wedge ms \leq \overline{ms}$ ⊃

$\langle\langle \overline{ms}, i\rangle, mi\rangle \xrightarrow{mt} \langle\langle (\overline{ms} - ms) | ms', i'\rangle, mit(f, mi)\rangle$ }

SMoLCS (BTS, ...) = MTS (PTS (STS (BTS, ...), ...), ...)

specfunct SLCS = λ spec BTS: IS-TS [STATE, FLAG, E_T, →]
spec INF, SFLAG
formales E_{SST}, E_{SYNC}, E_{PAR}
PTS (STS (BTS, INF, SFLAG, E_{SST}, E_{SYNC}), E_{PAR})

specfunct SMoLCS = λ spec BTS: IS-TS [STATE, FLAG, E_T, →]
spec INF, SFLAG, MINF, MFLAG
formales E_{SST}, E_{SYNC}, E_{PAR}, E_{MST}, E_{MON}
MTS (SLCS (BTS, INF, SFLAG, E_{SST}, E_{SYNC}, E_{PAR})[→/⇒]
MINF, E_{MST}, E_{MON})

- hierarchical SMoLCS
  SMoLCS (SMoLCS (. . .), . . .)
  ex. Cnet 2-level architecture

- inductive SMoLCS
  SYS = SMoLCS (BTS, . . .)
  SYS-state is BTS-state
  SYS-⇒ is BTS-→

Observational Semantics

TREEFLST = specification of finite trees with arcs labelled by flags and leaves labelled by states (lv: State → tree
.o.: flag × mset(tree) → tree)

specfunct TREE-SEM = λ spec TS: IS-TS [STATE, FLAG, E_T, →]
enrich TS + TREEFLST by
opns . ⇨ . : state × mset(tree) → bool
axioms { D(s) ⊃ s ⇨ lv(s) } ∪
{ ⋀ (i=1..n) (s -fi-> si ∧ si ⇨ mTi) ⋀ (i,j=1, i≠j) (fi,si) ≠ (fj,sj) ⊃
s ⇨ f1 . mt1 | . . . | fn . mtn | n ≥ 1 }

specfunct OBS-SEM = λ spec TS: IS-TS [STATE, FLAG, E_T, →]
spec OBS
formales E_{SEM}
derive
observe
enrich TREE-SEM(TS) + OBS by
opns obsmap: mset(tree) → obs
$\cdot \Rightarrow_{obs} \cdot$: state × obs → bool
axioms . .
by { $\Rightarrow_{obs}$ (s, x) | s ∈ $W_{sig(STATE)}$ ∧ x ∈ X_{obs} }
by insig (STATE ∪ OBS ∪ { $\Rightarrow_{obs}$ })

OBS
adt of what is visible of a tree

obsmap
- defined by E_{SEM}
- abstracts the visible part

$\Rightarrow_{obs}$
defines the set of contexts which we observe

ex. specfunct TO-SEM = λ spec TS: IS-TS [STATE, FLAG, E_T, →]
spec RESULT
OBS-SEM (TS, SET(RESULT), E_{10})

(RESULT contains
normal: state → bool
res: state → result)

Teoremi caratterizzanti le proprietà dei sistemi semantici

Foundations

Structured algebraic specifications: a kernel language
Hab. Thesis, München, 1983 (Wirsing)

Combining an operational with an algebraic spec. of concurrency
Workshop on Combining Spec Methods
Nyborg (DK), 1984 (Astesiano)

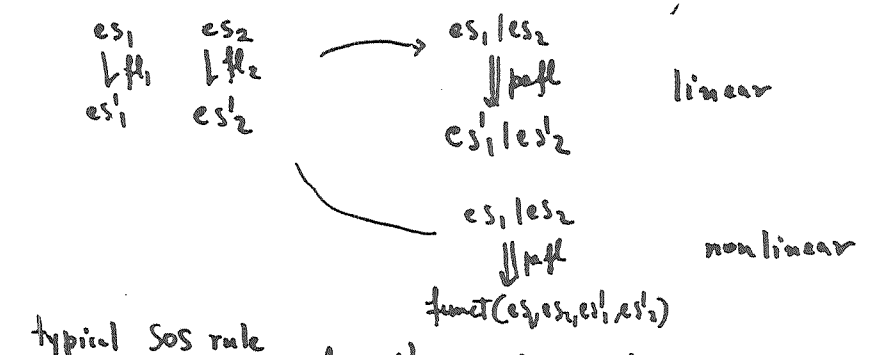
- On the parameterized algebraic spec of conc. syst.
CAAP-TAPSOFT'85, Berlin, 1985
(Astesiano, Navarri, Reggio, Wirsing)

Applications

- Specifications
  - Formal specification of a concurrent architecture in a real project
    (Astesiano, Mazzanti, Reggio, Zucca) 1984
    Cnet inter-node communication mechanisms
    Full report in preparation
- Properties of conc. syst.
  - A semantic analysis of fifo policies in abstract concurrent systems
    (Astesiano, Reggio) 1986
    relationship between queue handling mechanisms and fifo satisfaction of resource requests

Further steps

1. Hierarchical systems with non homogeneous entities
   - sum operation on entities
     (the component processes can be of very different nature - need only congruent interfaces)
   - hierarchy is permitted by modularity
     a system has the same nature of the component entities
     (i.e. it satisfies the same general axioms)
     [it is a SMo(L)CS]
2. Inductive systems
   CCS configuration = entity stage
   merging/renaming operation
   as-es : conf → es
3. Linear vs. nonlinear systems



typical SOS rule for parallel comb. mech.

$$\frac{f \to f' \quad g \to g' \quad x \to x'}{S\,f\,g\,x \to (f'x')(g'x')}$$

1. Can treat non lin. syst. (SMoCS) (ex.: every SOS par. syst → SMoCS syst.)
2. But linearization helps (is essential?) in discussing distributed control
3. Linearization conjecture

Tools ⟶ Aims

unique (linguistic) framework ⟶ specifying sw. architectures / languages at different levels

ex. CMET
- the internode comm. net
- the intranode comm. schemas (ADA-like)
- the system language (ADA+)
- the impl. code (X-code)

unique conceptual schema ⟶ studying basic concepts (synchronis., communic., parallelism, distributed control, interference ...)
- compare similar systems (in which some parts are the same)
  interleaving vs parallel
  distributed vs centralized

algebraic setting + parameterization ⟶ abstraction
- axiom. specif. of static configurations (edit style, not a fixed language)
- axiom. specif. of visibility conditions on transitions (to avoid, possibly, the "grain of actions" problem)

Results, problems & future work

1. Results on relationships between interleaving and parallel execution, timed / untimed systems, influence of queuing policies, etc.
   Attempt at defining precisely distribution (various levels of)

2. : "Grain of actions" problem (Lamport)
   a trans. syst. refers to a grain of action
   in SMoLCS we have a relative freedom, since we are not bound to a language
   nevertheless when we need to compare times, we reintroduce a grain of the actions
   ex.: Timed statements, bounded durations

   escape: 1. prove equivalence theorems
   2. specification of time conditions on actions and their composition
   ↓
   canonical transformation in SMoLCS

   related future work: extend 1 and 2. to express more general relationship between events
   (can see a connection with OLderog's work)

3. explore classes of equivalences produced by observability conditions
   relate them to classes of axiomatically defined models
   (as Hennessy-Milner, Stirling)

4. denotational semantics
   - should not be difficult extend and generalize Winskel's tree sem.
   don't know how to relate to De Bakker's sem.

SPECIFICHE DI APPLICAZIONI CHE USANO BASI DI DATI

ANTONIO ALBANO
Dipartimento di Informatica
Università di Pisa
Corso Italia, 40 - 56100 Pisa

Sommario

- Il problema
- Il Progetto Galileo
- I linguaggi Galileo e Galileo/R
- Conclusioni

☐ IL PROBLEMA

Noti

- i requisiti di più classi di utenti.
- le caratteristiche del DBMS.

Ricavare

- la descrizione della struttura logica della BD (*schema logico e schemi esterni*).
- la descrizione dell'organizzazione fisica della BD (*schema fisico*).
- i programmi per le applicazioni.

☐ ASPETTI DEL PROBLEMA

- Cosa si modella
- Con quali meccanismi di astrazione

  (*modello concettuale*)
- Come si procede

  (*metodologia di progettazione*)
- Con quali linguaggi e strumenti

☐ COSA MODELLARE

La conoscenza concreta,

cioè i fatti specifici che si vogliono rappresentare come dati: le entità, le loro caratteristiche e le associazioni fra entità.

La conoscenza astratta,

cioè i fatti generali che descrivono la conoscenza concreta rappresentata (*vincoli d'integrità*) e limitano il modo in cui essa può evolvere.

La conoscenza procedurale,

cioè i modi in cui si può operare sulla conoscenza concreta per modificarla o per ricavare altri fatti con un procedimento di calcolo.

La dinamica

cioe il modo in cui le attività dell'organizzazione vengono svolte utilizzando la conoscenza procedurale e come le attività interagiscono quando sono eseguite contemporaneamente.

La struttura,

cioè quali unità (sottosistemi) esistono nel sistema osservato e come interagiscono.

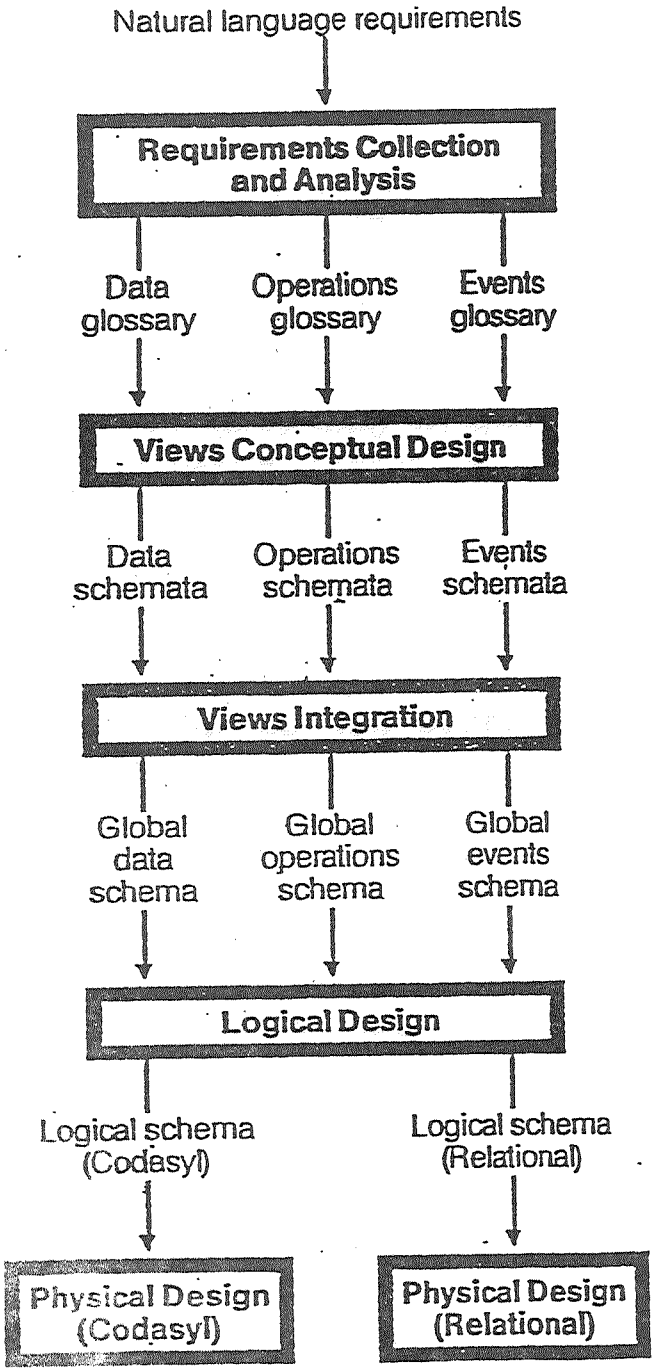
Le comunicazioni.

cioè le modalità offerte agli utenti per accedere alle risorse informative.

☐ *OSSERVAZIONE*

SVILUPPARE APPLICAZIONI CHE USANO BASI DI DATI SIGNIFICA SVILUPPARE DEL SOFTWARE. PERTANTO I METODI DI SPECIFICA PROPOSTI NELL'AREA DELL'INGEGNERIA INFORMATICA SI POSSONO IMPIEGARE ANCHE IN QUESTO SETTORE, CHE PERO' PRESENTA ALCUNI ASPETTI SPECIFICI CHE RENDONO NON BANALE IL TRASFERIMENTO DEI RISULTATI:

- La struttura logica complessa dei dati è l'aspetto dominante.
- Le applicazioni hanno il caso 'normale' semplice, ma le 'eccezioni' sono tante.
- I sistemi da realizzare sono interattivi.
- E' raro partire da requisiti stabili, pertanto occorre anticipare al massimo le fasi di validazione e prova.



Overall structure of the DATAID-1 methodology.

☐ *OSSERVAZIONE*

LA SPECIFICA DEI REQUISITI, IL PROGETTO CONCETTUALE E IL PROGETTO LOGICO VENGONO VISTI COME MODELLI DEL SISTEMA DA REALIZZARE, AD UN DIVERSO LIVELLO DI COMPLETEZZA:

DA UNA VISIONE PIU' AD ALTO LIVELLO, ORIENTATA AGLI UTENTI, CONCETTUALE, AD UNA VISIONE PIU' ORIENTATA ALLA MACCHINA

☐ *LIMITI DELL'APPROCCIO*

- Orientato sostanzialmente all'analisi dei dati, e al progetto logico e fisico della base di dati.

- Molta attenzione sulla rappresentazione grafica degli aspetti da specificare, con la speranza che ciò possa agevolare la validazione dei requisiti da parte dei committenti.

  ma poca attenzione alle possibilità di prova delle specifiche, perché date in un linguaggio non eseguibile.

- La codifica delle applicazioni rimane un processo manuale *estraneo* al processo di progettazione.

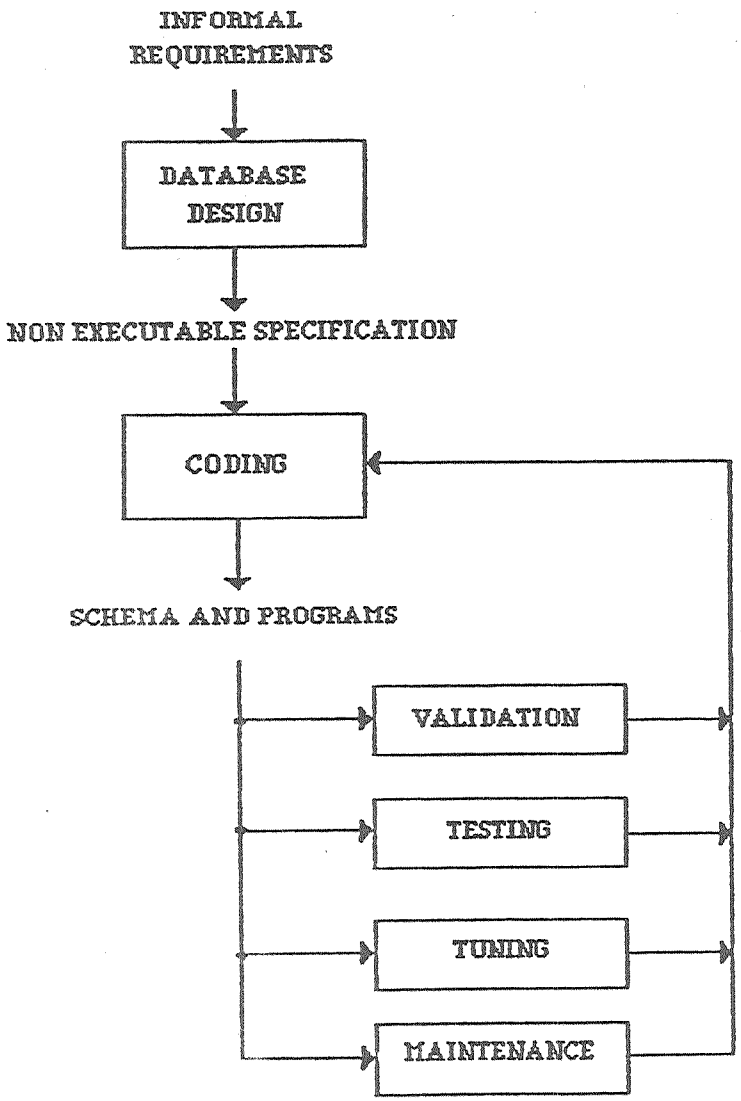
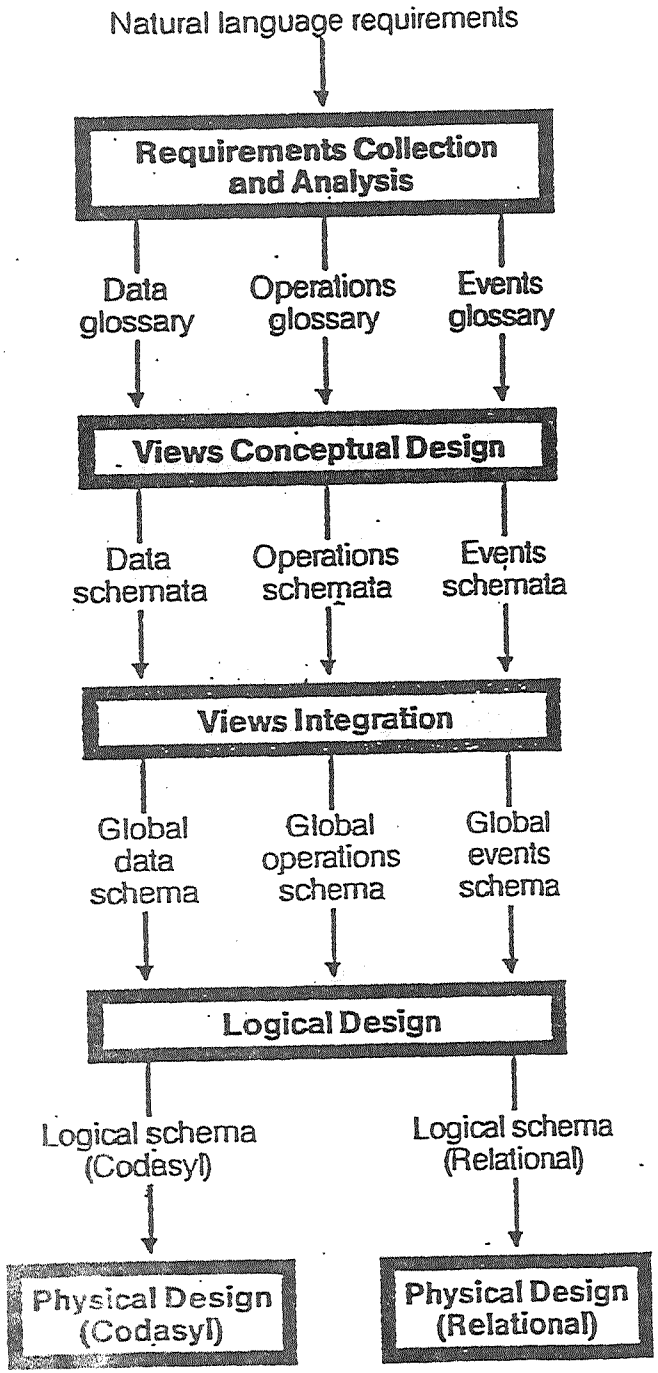


Fig. 1 CONVENTIONAL PARADIGM

PROGETTO Galileo

- Il linguaggio concettuale Galileo e le sue versioni: Gerarchica, Reticolare e Relazionale
- Il linguaggio per la specifica dei requisiti: Il Galileo/R
- Il Sistema Dialogo: un ambiente integrato di strumenti per la prototipizzazione e sviluppo di applicazioni per basi di dati



Overall structure of the DATAID-1 methodology.

☐ *OSSERVAZIONE*

LA SPECIFICA DEI REQUISITI, IL PROGETTO CONCETTUALE E IL PROGETTO LOGICO VENGONO VISTI COME MODELLI DEL SISTEMA DA REALIZ-ARE, AD UN DIVERSO LIVELLO DI COMPLETEZZA:

DA UNA VISIONE PIU' AD ALTO LIVELLO, ORIENTATA AGLI UTENTI, CONCETTUALE, AD UNA VISIONE PIU' ORIENTATA ALLA MACCHINA

☐ *LIMITI DELL'APPROCCIO*

- Orientato sostanzialmente all'analisi dei dati, e al progetto logico e fisico della base di dati.

- Molta attenzione sulla rappresentazione grafica degli aspetti da specificare, con la speranza che ciò possa agevolare la validazione dei requisiti da parte dei committenti.

  ma poca attenzione alle possibilità di prova delle specifiche, perché date in un linguaggio non eseguibile.

- La codifica delle applicazioni rimane un processo manuale *estraneo* al processo di progettazione.

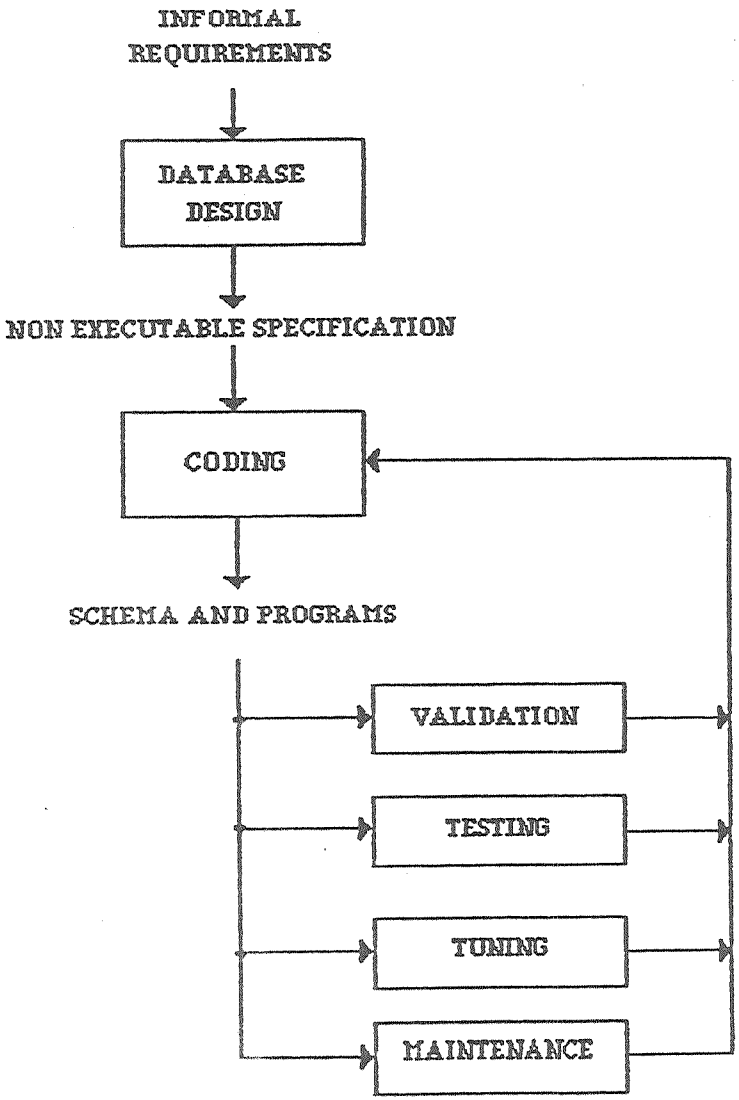


Fig. 1 CONVENTIONAL PARADIGM

PROGETTO Galileo

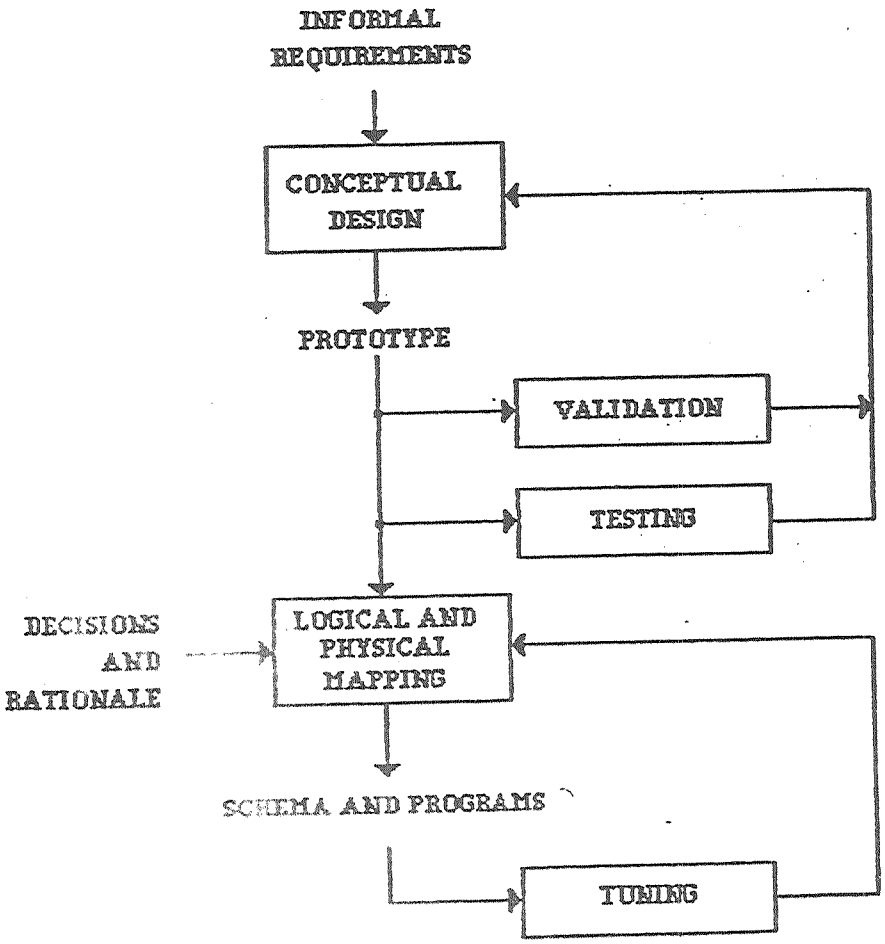
- Il linguaggio concettuale Galileo e le sue versioni: Gerarchica, Reticolare e Relazionale
- Il linguaggio per la specifica dei requisiti: Il Galileo/R
- Il Sistema Dialogo: un ambiente integrato di strumenti per la prototipizzazione e sviluppo di applicazioni per basi di dati

SCOPO DELLA PROGETTAZIONE CONCETTUALE

Realizzazione di un prototipo funzionante, al livello di definizione desiderato, di cio' che va realizzato su un sistema commerciale.

PROTOTIPO

- E' una versione semplificata, ma funzionante del sistema da realizzare.
- Il committente deve fare esperimenti con il prototipo per decidere se rivedere i requisiti.
- Il prototipo deve essere poco costoso da costruire.
- Il passaggio dal prototipo al sistema funzionante deve essere meccanizzato.
- Il prototipo non e' del tipo "usa e getta", ma e' l'oggetto della manutenzione.



ERATIONAL PARADIGM

CARATTERISTICHE DEL Galileo:

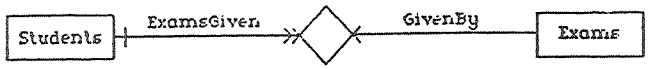
- E` un linguaggio concettuale di tipo funzionale, interattivo, fortemente tipizzato, con legami statici
- Supporta i meccanismi di astrazione dei Modelli Semantici dei Dati e dei Linguaggi di Programmazione
- Ha gerarchie fra tipi
- Ha un meccanismo di strutturazione per organizzare descrizioni complesse di basi di dati in unita` interconnesse
- Ha un meccanismo per il trattamento delle eccezioni
- E` orientato alle transazioni
- Ha un meccanismo di processi

DIFFERENZE DALL'Edinburgh ML:

- Diversa modalita' di definizione dei tipi astratti
- Tipi con asserzioni
- Gerarchie fra tipi
- Classi
- Ambienti
- Transazioni
- Astrazione dalla persistenza dei dati
- Processi

```
E: use type Month <=> int assert with "Illegal Month"
                              this within (1,12)
=  with newMonth (M : int) : Month := mkMonth (M)
=  and NextMonth (M : Month) : Month :=
=         use ThisMonth := repMonth (M)
=       ) in if ThisMonth < 12
=            then mkMonth (ThisMonth + 1)
=            else mkMonth (1)
=  and EqMonth (M : Month, N : Month) : bool :=
=         repMonth (M) = repMonth (N);

>  newType Month = -
|  NewMonth = fun : int -> Month
|  NextMonth = fun : Month -> Month
|  EqMonth = fun : Month, Month -> bool
```

```
E: use type AnAge <-> int assert this within (1,150);

> newType AnAge = -
| mkAnAge = fun : int -> AnAge
| repAnAge = fun : AnAge -> int
| > = fun : AnAge, AnAge -> bool
| < = fun : AnAge, AnAge -> bool
| = = fun : AnAge, AnAge -> bool
| <> = fun : AnAge, AnAge -> bool
| >= = fun : AnAge, AnAge -> bool
| <= = fun : AnAge, AnAge -> bool
| ~ = fun : AnAge -> AnAge
| + = fun : AnAge, AnAge -> AnAge
| - = fun : AnAge, AnAge -> AnAge
| * = fun : AnAge, AnAge -> AnAge
| div = fun : AnAge, AnAge -> AnAge
| mod = fun : AnAge, AnAge -> AnAge

E: use MyAge = mkAnAge (39):

> MyAge = - : AnAge

E: MyAge + MyAge;

  - : AnAge

E: MyAge + 1;

  Type clash in : MyAge + 1
  Looking for : AnAge
  I have found : Int

E: use type Date <-> (Day : int assert this within (1,31)
=                     and Month : int assert this within (1,12)
=                     and Year : int assert this within (1983, 2000))
=                    assert with "Illegal Date"
=                    use this
=                     in Day ≤ (if Month = 2
=                               then if Year mod 4 = 0 then 29 else 28
=                               else if Month=4 or Month=6 or Month=9
=                                       or Month=11
=                                    then 30 else 31)

> newType Date = -
| mkDate = fun : (Day : int and Month : int and Year : int ) -> Date
| repDate = fun : Date -> (Day : int and Month : int and Year : int )

E: use Today := var mkDate (Day := 19 and Month := 9 and Year :=
1984);

> Today = var - : var Date

E: use type Person <->
=       (Age := derived use this
=                    in (Year of at Today - Year of BirthDate)
=        and BirthDate : Date
=        and Citizenship : default "Italian"
=        and Name : string
=        and Phone : optional int assert
=                                  this within (100000,999999));

> newType Person = -
| mkPerson = fun : (Age: default int and BirthDate: Date
|                   and Citizenship: string
|                   and Name: string and Phone: int)
|                   -> Person
| repPerson = fun : Person ->
|                   (Age: default int and BirthDate: Date
|                   and Citizenship: string and Name: string
|                   and Phone: int)

E: use Smith := mkPerson (Name := "Henry Smith"
=                         and BirthDate := mkDate (Year := 1945
=                                                  and Month := 3
=                                                  and Day := 1));

> Smith : Person
```




```
E: use
=  rec type PhoneNumber <-> int assert this within (100000,999999)
=  and type SixDigits <-> int assert this within (100000,999999)
=  and Students class
=        Student <->
=          (BirthDate : Date
=           and Citizenship : default "Italian"
=           and ExamsGiven : derived all Exams with GivenBy = this
=           and Name : string
=           and StudentNumber : SixDigits
=           and Phone : optional PhoneNumber);
=           key (StudentNumber)
=  and Exams class
=        Exam <->
=          (GivenBy : Student
=           and Course : string
=           and Date : Date
=           and Grade : int assert this within(1,100)
=           and Honor : <Yes or No>
=           ext StudentNumber := derived StudentNumber of GivenBy)
=           key (GivenBy,Course)
=           assert with "IllegalHonor"
=                    if Honor of this is Yes then Grade of this = 100
=                                            else true;

>  newType PhoneNumber = -
|  newType SixDigits = -
|  class Students = - : seq Student
|  newType Student = -
|  class Exams = - : seq Exam
|  newType Exam = -
```

```
E: mkStudent (StudentNumber := mkSixDigits (222222)
=             and BirthDate :=
=                   mkDate (Year := 1963
=                           and Month := 5
=                           and Day := 10)
=             and Name := "Paul Smith")

   - : Student

E: mkExam (GivenBy := get Students
=                           with StudentNumber = mkSixDigits (222222)
=          and Course := "CS3"
=          and Date := mkDate (Year := 1983
=                              and Month := 5
=                              and Day := 10)
=          and Grade := 89
=          and Honor := <No>);

   - : Exam
```

IL GALILEO/R

IPOTESI

- I requisiti vanno specificati a diversi livelli di dettaglio e precisione.

- L'analista deve dare un modello del comportamento aspettato del sistema.

- Nella descrizione dei requisiti e' importante usare meccanismi d'astrazione espressivi.

- Per facilitare la progettazione concettuale e' bene che i meccanismi di astrazione siano gli stessi.

COSA DESCRIVERE

- Settori aziendali ed interfacce

- I dati e le operazioni

- Vincoli d'integrita'

- Le operazioni sui dati

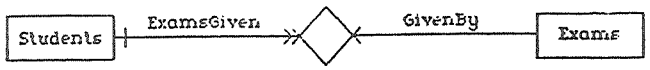
- Le attivita', cioe' come utilizzare i dati e le operazioni per espletare i compiti aziendali, e le modalita' di attivazione e di comunicazione.

- Parametri quantitativi

- Tipo di sicurezza dei dati

- Prestazioni attese dal sistema

COME PROCEDERE




```
E: use
=  rec type PhoneNumber <-> int assert this within (100000,999999)
=  and type SixDigits <-> int assert this within (100000,999999)
=  and Students class
=        Student <->
=          (BirthDate : Date
=           and Citizenship : default "Italian"
=           and ExamsGiven : derived all Exams with GivenBy = this
=           and Name : string
=           and StudentNumber : SixDigits
=           and Phone : optional PhoneNumber);
=           key (StudentNumber)
=  and Exams class
=        Exam <->
=          (GivenBy : Student
=           and Course : string
=           and Date : Date
=           and Grade : int assert this within(1,100)
=           and Honor : <Yes or No>
=           ext StudentNumber := derived StudentNumber of GivenBy)
=           key (GivenBy,Course)
=           assert with "IllegalHonor"
=                    if Honor of this is Yes then Grade of this = 100
=                                            else true;

>  newType PhoneNumber = -
|  newType SixDigits = -
|  class Students = - : seq Student
|  newType Student = -
|  class Exams = - : seq Exam
|  newType Exam = -
```

```
E: mkStudent (StudentNumber := mkSixDigits (222222)
=             and BirthDate :=
=                   mkDate (Year := 1963
=                           and Month := 5
=                           and Day := 10)
=             and Name := "Paul Smith")

   - : Student

E: mkExam (GivenBy := get Students
=                           with StudentNumber = mkSixDigits (222222)
=          and Course := "CS3"
=          and Date := mkDate (Year := 1983
=                              and Month := 5
=                              and Day := 10)
=          and Grade := 89
=          and Honor := <No>);

   - : Exam
```

IL GALILEO/R

IPOTESI

- I requisiti vanno specificati a diversi livelli di dettaglio e precisione.

- L'analista deve dare un modello del comportamento aspettato del sistema.

- Nella descrizione dei requisiti e' importante usare meccanismi d'astrazione espressivi.

- Per facilitare la progettazione concettuale e' bene che i meccanismi di astrazione siano gli stessi.

COSA DESCRIVERE

- Settori aziendali ed interfacce

- I dati e le operazioni

- Vincoli d'integrita'

- Le operazioni sui dati

- Le attivita', cioe' come utilizzare i dati e le operazioni per espletare i compiti aziendali, e le modalita' di attivazione e di comunicazione.

- Parametri quantitativi

- Tipo di sicurezza dei dati

- Prestazioni attese dal sistema

COME PROCEDERE

```
Students class

   description  « Facts about students of interest in this
                  application »
   user RegistrationOffice
   cardinality min = 200
               max = 3000
               avg = 1500
   inserted by EnrollStudent
   referenced by TakeOutBook
   updated by ANewExam quantity 1 using Exams
   Student <->
      (BirthDate : Date
       and Citizenship : default "Italian"
       and Name : string
       and StudentNumber : SixDigits
       and Phone : optional PhoneNumber
           description « phone number with format ddd-dddd »
       and Exams :
           association
           invert Exams on GivenBy
           cardinality
                      min = 0
                      max = 20
                      avg = 10
       and BooksBorrowed :
           association derived all Books with LoanedTo = this
   assert « a student cannot give an exam more than once »
   keys (StudentNumber)
```

```
TakeOutBook :=
   operation ( AStudentNumber : SixDigits,
               ACallNumber : BookNumber)
   description « To give a book on loan to a student »
   user Librarian
   used in activity Loan

   pre-condition
      description
                « A student exists with the specified StudentNumber,
                  a book exists with the specified CallNumber, and
                  the book is not on loan »
      (exactly 1 Students with StudentNumber = AStudentNumber
       And exactly 1 Books with CallNumber = ACallNumber
       And LoanedTo of (get Books with CallNumber = ACallNumber)
           is unbound)

   post-condition
      (LoanedTo of (get Books with CallNumber = ACallNumber) =
        (get Students with StudentNumber = AStudentNumber))

   exceptions
      if Not exactly 1 Students with StudentNumber = AStudentNumber
         then failwith "Student Unknown"
      if Not exactly 1 Books with CallNumber = ACallNumber
         then failwith "Book Unknown"
      if LoanedTo of (get Books with CallNumber = ACallNumber)
          is bound
         then failwith "Book on loan"

   class-used
      retrieves Students quantity 1 using StudentNumber
      retrieves Books quantity 1 using CallNumber
      updates Books quantity 1 using LoanedTo

   association-used
      modifies LoanedTo from Books to Students

   actions
```

```
activity Loan (AStudentNumber : SixDigits, ACallNumber : BookNumber)

   description « this is the requirements specification of the
                 process described previously in Galileo »
   modality on-line
   priority low
   happens 50 times-per day
   invokes Alarm
   employes TakeOutBook
   references Students Books

   messagges
      send Terminate to Alarm
      send AlreadyOnLoan to mycreator
      send NoMoreRenewals to mycreator
      receive TimeOut from Alarm
      receive Renewal from mycreator
      receive BookReturned from mycreator

   process
      if « the book is not available »
      then send AlreadyOnLoan to mycreator
      else (TakeOutBook (AStudentNumber, ACallNumber);
            « activate Alarm for loan duration »;
            use Returned := var false
            in while Not at Returned do
                 alternative
                    receive TimeOut() from Alarm
                     in « send a solicitation to the student and
                          restart Alarm »
                 or receive Renewal() from mycreator
                     in « if he has not yet done three renewals,
                           accept it and reset Alarm »
                 or receive BookReturned() from mycreator
                     in (Returned <- true;
                         « make book available »
```

☐ CONCLUSIONI

☐☐ Galileo

- Compilatore e interprete integrato con editore.
- Controllo asserzioni su valori modificabili
- Gestione della persistenza dei valori
- Interazione grafica per l'uso della base di dati
- Trasformazioni del prototipo in Galileo in programmi per DBMS.

☐☐ Galileo/R

- Interazione grafica per la specifica dei requisiti
- Controlli sulle specifiche

☐☐ Sperimentazioni ed integrazione degli strumenti nell'ambiente di sviluppo Dialogo.

SPECIFICHE BASATE SULLA LOGICA

DOTT. MAURIZIO MARTELLI

La Logica Matematica ha avuto, fin dalle origini, l'obiettivo di essere:

UN LINGUAGGIO UNIVERSALE PER RAPPRESENTARE IN MODO SISTEMATICO E MATEMATICO OGNI POSSIBILE FORMA DEL PENSIERO RAZIONALE, CONDENSABILE IN UN RAGIONAMENTO DEDUTTIVO.

⇓

Buoni candidati per specificare problemi, per rappresentarli formalmente e per ragionarci sopra.

Infatti:

- LINGUAGGIO NON-AMBIGUO
- LINGUAGGIO GRAMMATICALMENTE SEMPLICE
- LINGUAGGIO SIMBOLICO E FORMALE

Cosa usare della logica?

CALCOLO PROPOSIZIONALE (troppo limitato)

LOGICA DEL 1° ORDINE (adatta alla specifica ma non facilmente trattabile in modo automatico).

Come usare la logica?

ASSUNZIONI per descrivere il problema

CONCLUSIONI che possono essere tratte dalle assunzioni

CIOÈ: AUTOMATIZZARE IL PROCESSO DEDUTTIVO (COSTRUIRE PROVE A PARTIRE DA DEGLI ASSIOMI)

Soluzione:

FORMA CLAUSALE
(forma particolare della Logica del 1° ordine)

Cosa posso esprimere?

FRASI ATOMICHE (asserzioni di relazioni tra individui)

Carlo ama il teatro
Piero ha 2 anni più di Maria
⋮

FRASI COMPLESSE (implicazioni)

Maria ama Luca [se] Luca ama Maria
A Piero piace x [se] a x piace la logica
A x piace y [se] x è una donna [e] y è una donna
⋮ [e] y è bella

In generale la forma a clausole è del tipo:

$A_1 \boxed{o} A_2 \boxed{o} \dots \boxed{o} A_n \boxed{se} B_1 \boxed{e} B_2 \boxed{e} \dots \boxed{e} B_m$

(la verità di B_1, B_2 e B_m implicano la verità di A_1 o A_2 o ... A_n)

A_i e B_j sono formule atomiche:

costrutti linguistici che definiscono relazioni tra:

individui (costanti) [Carlo, il teatro, ...

variabili [x, ...

strutture (simboli di funzione o costrutti che servono a costruire oggetti più complessi)

[es.: il padre di Mario (individuo e non relazione)

LA FORMA CLAUSALE È ABBASTANZA POTENTE PER RAPPRESENTARE DIRETTAMENTE I PROBLEMI, SENZA CIOÈ DOVER ADOPERARE LA LOGICA DEL 1° ORDINE.

LOGICA

TUTTI I QUESITI RIGUARDANTI L'IMPLICAZIONE LOGICA NEL CALCOLO DEI PREDICATI DEL PRIMO ORDINE POSSONO ESSERE RIMPIAZZATI DA QUESITI RIGUARDANTI LA NON SODDISFACIBILITÀ DI <u>FORMULE IN FORMA CLAUSALE</u>:

INSIEME DI <u>CLAUSOLE</u>

CLAUSOLA: $B_1, \dots, B_m \leftarrow A_1, \dots, A_n$

A_i e B_i sono <u>FORMULE ATOMICHE</u> del tipo

$P(t_1, \dots, t_k)$

P è un predicato k-arie e i t_i sono <u>termini</u>

TERMINE < VARIABILE o COSTANTE
SIMBOLO DI FUNZIONE APPLICATO A TERMINI

<u>SEMANTICA</u> di un insieme di clausole S

$S = \{C_1, \dots, C_m\}$ è una congiuntione di clausole: $C_1 \wedge C_2 \wedge \dots \wedge C_m$

ogni clausola $B_1, \dots, B_m \leftarrow A_1, \dots, A_n$ è

$\forall x_1, \dots, x_k \, (B_1 \vee B_2 \vee \dots \vee B_m \leftarrow A_1 \wedge A_2 \wedge \dots \wedge A_n)$

o anche $\forall x_1, \dots, x_k \, (B_1 \vee B_2 \vee \dots B_m \vee \neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_n)$

casi particolari

$n = 0$ $\forall x_1, \dots, x_k \, (B_1 \vee B_2 \vee \dots B_m)$ (assertione)

$m = 0$ $\forall x_1, \dots, x_k \, \neg (A_1 \wedge A_2 \dots \wedge A_n)$ (goal)

$n = m = 0$ $\square$ <u>CLAUSOLA NULLA</u> (contraddizione)

LE CLAUSOLE HORN

FORMA RISTRETTA CON AL PIÙ UNA FORMULA ATOMICA A SINISTRA DELLA ←

ASSERZIONI	(A ←)
REGOLE	(A ← B, C)
GOAL	(← A, B)

Possibili interpretazioni di una clausola

A ← B, C

- DICHIARATIVA: A è vera se B e C sono veri
- PROCEDURALE: per risolvere il problema A riducilo ai sottoproblemi B e C

Caratteristiche:

- non permettono di rappresentare qualsiasi problema rappresentabile nella logica del 1° ordine
- ma permettono di rappresentare tutte le funzioni calcolabili.
- hanno procedure di prova basate sulle risoluzione che risultano essere efficienti e che si basano sulla negazione di un certo fatto (goal) e sulla ricerca di una contraddizione nella teoria composta dagli assiomi (assertioni e regole) più il goal (dimostrazione per assurdo).

METODO DI SPECIFICA ASSIOMATICO

PROBLEMA DELLO STACK:

La struttura stack è rappresentata da:

- empty
- stack(x,s)

(stack è un costruttore di dato)

OPERAZIONI:

Push (s, x, Stack(x,s)) ←

Pop (empty, errore) ←

Pop (Stack (x,s) , s) ←

Top (empty, errore) ←

Top (Stack (x,s), x) ←

Per controllare il funzionamento della specifica (eseguibile) posso saivere vari goal

← Push(empty, a, S), Pop (S, empty)

ma passare da un problema alla specifica? (*

<u>es esempio</u>:

linguaggio naturale →
ling. naturale preciso e non ambiguo →
→ ling. formale.

---

esempio 1: PROBLEMA

Data una sequenza di n interi, una <u>su-sequenza</u> è una sottosequenza ordinata in ordine crescente. Una <u>sottosequenza</u> è qualsiasi sottinsieme della sequenza originale dove l'ordine originale è mantenuto (vi sono 2^n possibili sottosequenze). Ordinata in ordine crescente significa che nessun elemento della su-sequenza ha alla destra un numero minore.
Esprimere lo sviluppo di un algoritmo che, data una sequenza, calcoli la lunghezza della sua più lunga su-sequenza.
Notare che tutte le sottosequenze di lunghezza 1 sono su-sequenze.

Es: (3, 1, 1, 2, 5, 3) → (1, 1, 2, 5) → (4)
(3, 1, 1, 2, 5, 3) → (1, 1, 2, 3) → (4)

PROBLEMA (2): (

Data una lista L, trovare n (lunghezza della più lunga su-sequenza)

a) se L è vuota n è 0.
b) se L ha un solo elemento n è 1.
c) se L ha più di un elemento, sia A la testa di L e B la coda.
Il numero n sarà il massimo tra 2 numeri m e h ottenuti rispettivamente considerando:
- tutte le su-sequenze contenenti A (*)
- tutte le su-sequenze senza A (+)

[si sfrutta il fatto che le sottosequenze sono 2^n]

(+) è il problema originario con lista B e numero h

(*) è un sottoproblema:

Dato un elemento A, una lista B, trovare il numero n (lunghezza della più lunga su-sequenza che inizia con A).

d) se B è vuota n è 1
e) se B non è vuota, sia C la testa di B e D la coda.
e1) se A < C, C può essere scartato e si ripete il problema (*) con A e la lista D.
e2) se A ≥ C, avremo ancora 2 casi: n sarà il massimo tra m e h ottenuti rispettivamente considerando la su-seq. con A e C (-) e con A e senza C (!).

(-) è il problema (*) con elemento C e lista D e lunghezza m-1 → m = m+1. (c'è A in più).

(!) è il problema (*) con elemento A e lista D.

PROBLEMA (3)

Lista [ ]
[A|B]

Problema Sub(L,N)

a) Sub([ ],0) ←
b) Sub([A],1) ←
c) Sub([A|B],N) ← max(M,N,H),
Sub1(A,B,M), (*)
Sub(B,H) (+)

Problema (*) Sub1(A,B,N)

d) Sub1(A,[ ],1) ←
e1) Sub1(A,[C|D],N) ← A < C,
Sub1(A,D,N)

e2) Sub1(A,[C|D],N) ← A ≥ C,
max(M,H,N),
Sub1(C,D,M1), } (-)
M = M1 + 1 ,
Sub1(A,D,H) (!)

max(A,B,A) ← A ≥ B.
max(A,B,B) ← A < B.

Specifiche per Banche Dati e/o Regolamenti ?

Esempio 1:

GESTIONE BIBLIOTECA:

1) Un modulo separato deve essere compilato dall'utente per ogni volume preso in prestito.

2) I libri devono essere riportati entro la data di scadenza del prestito.

3) Gli utenti non devono superare un numero massimo consentito di libri in prestito.

4) Nessun libro sarà dato in prestito a chi ha libri da restituire dei quali la data di restituzione è scaduta

Quantità libri permessi:
Studenti 6
Dottorandi 10
Insegnanti 20

come immagazzinare l'informazione?

Persone : Carlo, Bruno, Luca, Sara, Sig.ra Bianchi ...

Libri : Logic for Problem Solving,
Mathematical Logic,
⋮

ASSERZIONI

Studente (Carlo).
DoHolando (Bruno).
Insegnante (Luca).
Bibliotecaria (Sig.ra Bianchi).
Libro (Logic ....).
Libro (Mathematical ...).

REGOLE GENERALI

Persona (x) ∃e Studente (x)
Persona (x) ∃e DoHolando (x)
Persona (x) ∃e Insegnante (x)
Persona (x) ∃e Bibliotecaria (x)

REGISTRAZIONE DI FATTI RELATIVI ALLA GESTIONE:

Sig.ra Bianchi: ha prestato Logic for Pr. Solv.
a Carlo il 1 Marzo M[illesimo]
⋮
Luca ha restituito Math. L. il 4 Nov. 1966.
⋮
La Porta deve essere restituito il 30 Nov. 1966.
⋮

Regole relative alla gestione

x ha y altempo T ∃e Bibliotecaria (z) ∈
z ha prestato y a x il T1 ∈
T ≥ T1 ∈
non esiste T' : (x ha restituito y il T' ∈ T1 ≤ T' ≤ T)

tempo scaduto per x il T ∃e
x deve essere restituito il T1 ∈
T ≥ T1

x ha un libro scaduto il T ∃e
x ha y altempo T ∈
tempo scaduto per x il T

DOMANDE AL SISTEMA:

Possibile (x, prendere (y), T) [SE]

Persona (x) [E]
Libro (y) [E]
[non esiste] z: z ha y al tempo T

x ha il permesso di prendere y al tempo T [SE]

Persona (x) [E]
[non] Troppi_libri (x, T) [E]
[non] x ha un libro scaduto al T

Troppi_libri (x, T) [SE]

Libri_presenti (x, N) [E]
Numero di libri (x, N, T)

Libri_presenti (x, 0) [SE] Studente (x)
Libri_presenti (x, 10) [SE] Dottorando (x)
Libri_presenti (x, 20) [SE] Impiegato (x)

La logica permette di descrivere solo problemi in modo inefficiente?

Vecchia critica

Problema:
Stabilire se un vettore X è ordinato

1) X è ordinato se per ogni $i, j (x_i \leq x_j$ se $i < j)$

2) X è ordinato se per ogni $i (x_i \leq x_{i+1})$

1) e 2) sono entrambi: specifiche

1) e 2) prefigurano algoritmi diversi

Se n è il numero degli elementi
1) è di ordine n^2
2) è di ordine n

PROLOG CORE LING. DI PROGRAMMAZIONE

di alto livello e orientato alla SPECIFICA
di piccole dimensioni (vedi consigli di HOARE Pascal e non Algol 68 o PL/1)

possibilità di prove di correttezza (teoremi in logica)

ling. basato sulle esigenze dell'utente e non su quelle della macchina (vedi consigli di Backus - architetture NON - VON NEUMANN)

Principio: (KOWALSKI)
Algoritmo = Logica + Controllo
↓ ↓
Dichiaratività operazionalità

APPLICAZIONI:

+ elaborazione del linguaggio naturale
+ dimostrazione automatica di teoremi
+ Data Base deduttivi
+ robotica
+ CAD e progettazione architetturale
+ risoluzione di equazioni simboliche
+ Sistemi Esperti
⋮
+ linguaggio di programmazione per bambini
+ analisi di programmi assembler (IBM)
+ descrizione e gestione di aspetti legali

CLAUSOLA HORN

A se F. e C (dichiarativo)

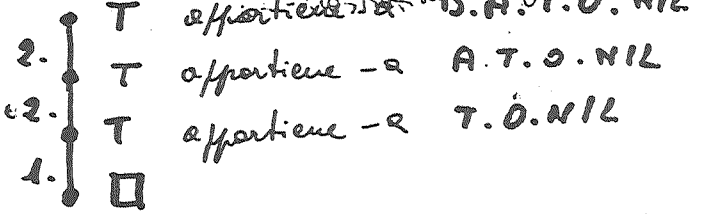
per risolvere A riduci a B e C (procedurale)

PROCEDURE DI PROVA BASATE SULLA RISOLUZIONE
E I COMPORTANO COME ESECUTORI DI PROGRAMMI

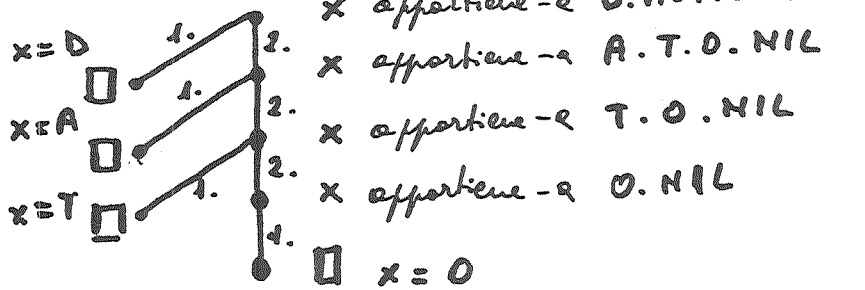
Es:
1. x appartiene-a x.y
2. x appartiene-a z.y se x appartiene-a y

1. asserzione 2. regola $x \cdot y = \overset{x}{\cdot}\overset{\cdots\cdots}{\cdot}\overset{y}{\cdot}$ (lista)

Se si vuole provare che 1. e 2. implicano qualcosa si può procedere all'indietro a partire dalla conclusione.



Se si vuole calcolare qualcosa:



(Backtracking e Non-determinismo)

0.0, S(x) = successore (x)

ESEMPIO

1. FATT(0, S(0)).
2. FATT(S(x), u) ← FATT(x, v), PROD(S(x), v, u)

Programma per calcolare il fattoriale.
Se vogliamo calcolare qualcosa, bisogna aggiungere il cosiddetto GOAL

3. ← FATT(S(S(0)), Y)

Aggiungere il GOAL equivale ad effettuare una dimostrazione per assurdo:
nessun x è il fattoriale di 2
3. contraddice 1. 2. che implicano logicamente che il fattoriale di 2 è 2.

LA PROCEDURA DI PROVA TROVA LA CONTRADDIZIONE (□) TROVANDO ANCHE UNA CONTRO-ISTANZA DI x CHE È PROPRIO S(S(0)).

CLAUSOLA HORN

$$B \leftarrow A_1, \dots, A_m$$

INTERPRETAZIONE PROCEDURALE

1. $B \leftarrow A_1, \dots, A_m$ — DICHIARAZIONE DI PROCEDURA
   (nome) (corpo)
2. A_i — CHIAMATA DI PROCEDURA
3. $\leftarrow A_1, \dots, A_m$ — (GOAL) PROGRAMMA PRINCIPALE
4. $B \leftarrow$ — ASSERZIONE - PROCEDURA SENZA CORPO
5. □ — ISTRUZIONE DI HALT

RISOLUZIONE

È IL MECCANISMO CON CUI VIENE EFFETTUATA LA CHIAMATA DI PROCEDURA:

da $\leftarrow A_1, \dots, A_{i-1}, A_i, A_{i+1}, \dots, A_m$

e $B \leftarrow B_1, \dots, B_m$

se qualche sostituzione di termini a variabili θ rende uguali A_i e B la risoluzione deriva il nuovo GOAL:

$$\leftarrow (A_1, \dots, A_{i-1}, B_1, \dots, B_m, A_{i+1}, \dots, A_m)\theta$$

ESEMPIO DI VALUTAZIONE

```
                ← Fatt(S(S(0)), x)
2.              ← FatH(S(0), v), Per(S(S(0)), v, x)
2.              ← FatH(0, v'), Per(S(0), v', v), Per(S(S(0)), v, x)
1.              ← Per(S(0), S(0), v), Per(S(S(0)), v, x)
T               ← Per(S(S(0)), S(0), x)
T               □        {x = S(S(0))}
```

```
1. FatH(0, S(0)).
2. FatH(S(x), u) ←
       FatH(x, v)
       Per(S(u), v, u).
```

COMPUTAZIONE

sequenza di goal

- SUCCESSO se □
- INSUCCESSO se una chiamata non può corrispondere a nessuna procedura

LOGICA

derivazioni via risoluzione

- ogni goal è conseguenza logica del predente e se si arriva a □ (contraddizione), questa è una prova di REFUTAZIONE, cioè di non soddisfacibilità -

GARANZIE ?

TEOREMI DI COMPLETEZZA E CORRETTEZZA

> Un insieme di clausole Horn S è non soddisfacibile se e solo se il sistema di inferenza ammette una refutazione di S

ESEMPIO

1. $+(0, x, x)$
2. $+(S(x), y, S(z)) \leftarrow +(x, y, z)$

SOMMA

$\leftarrow +(V_1, V_2, x)$

SOTTRAZIONE

$\leftarrow +(x, V_1, V_2)$

SCOMPOSIZIONE

$\leftarrow +(x, y, V)$

- CONCETTO DI INVERTIBILITA'
- POSSIBILITÀ DI CALCOLI NON-DETERMINISTICI

ASPETTI IMPORTANTI PER LA SPECIFICA

Cosa abbiamo oggi ?

PROLOG :

buon ling. di programmazione
ma non ancora ben attrezzato
come ling. di specifica

Ambienti
Costrutti Linguistici

Sviluppi promettenti :

- Tipi
- Concorrenza
- Query the user (Key)
- Ambienti di sviluppo
- Moduli

```
(1) module list_handling.
    MODULE list_handling.

(2) enter length(nil,0).
    length / 2

(3) e length(X.L,N) :- lengh(L,N1), N is N1 + 1.
    length / 2 + 2

(4) type
    length(X . L,N) :-
        lengh(L,N1), N is N1 + 1 .

(5) ? length(1.2.nil, N) .
    Exception -505: undefined predicate
      In call of lengh(2 . nil,_104)
    FUNCTION (h for help)?

(6) b
    lengh(2 . nil,_104)
    length(1 . 2 . nil, _93)
    FUNCTION (h for help)?

(7) a
    Execution aborted

(8) edit
     10:  length(X . L,N):-
     20:      lengh(L,N1),
     30:      N is N1 + 1
    *** Enter editor commands ***

(9) 20:       length(L,N1),
    *** Line 20 replaced ***

(10) end
    CLAUSE length / 2 + 2 REPLACED.

(11) ? length(1.2.nil,N).
          N = 2
    CONTINUE? (y/n)

(12) save myfile
    SAVING list_handling
                length / 2
    SAVED
```

Fig. 1. A dialogue with PDSS (first part)

```
(1) read myfile
    myfile READ.

(2) export length/2.
    EXPORT.

(3) mode length(+,–).
    MODE.

(4) type MODULE
    module list_handling.
    export (length / 2).
    /*$eject*/
    body.
    mode length(+,–).
    length (nil,0) .
    length (X . L, N) :–
        length (L,N1), N is N1 + 1 .
    endmod /* list_handling */ .

(5) save myfile
    SAVING list_handling
              FACES...
              length / 2
    SAVED
```

Fig.2. A dialogue with PDSS (second part)

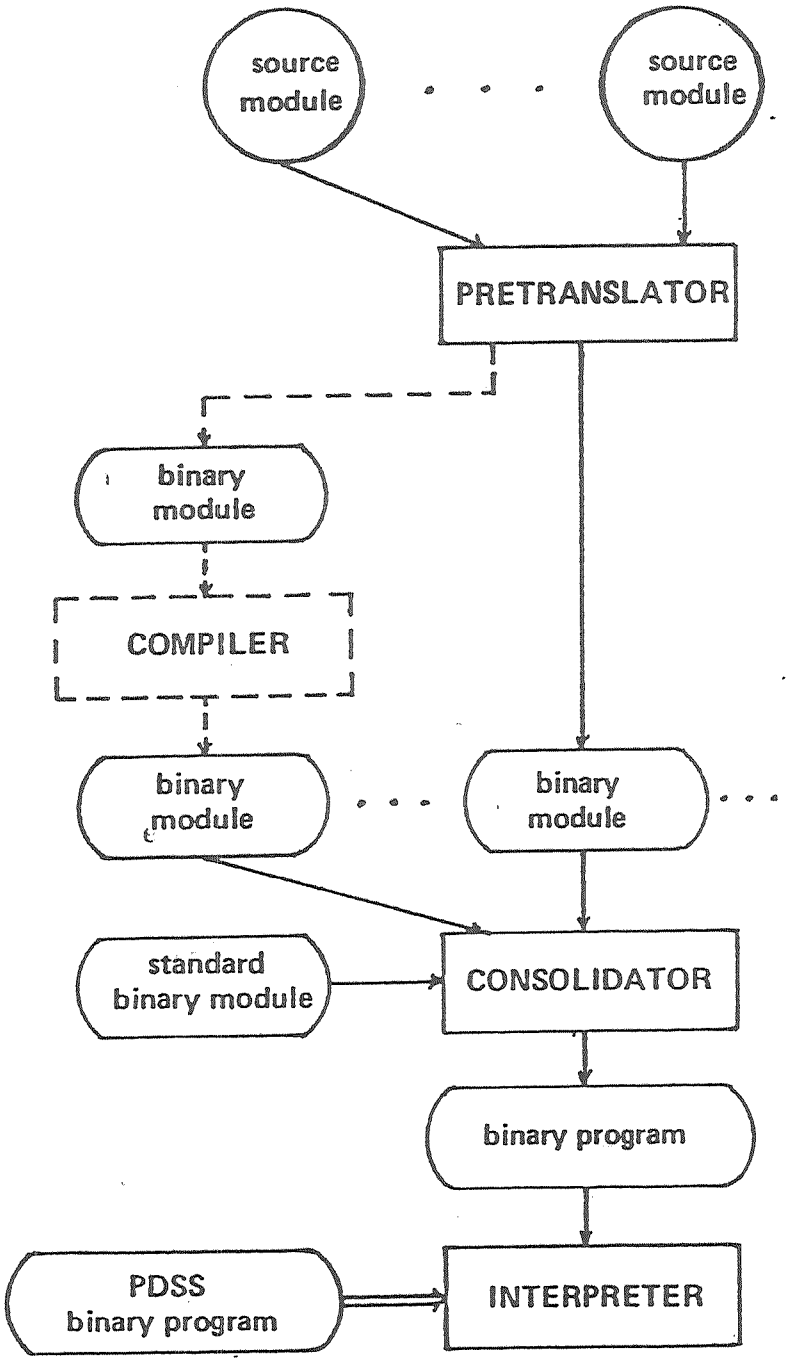


Fig. 3. The structure of the MPROLOG system

Futuro ?

- Integrazione tra logic e functional programming
  - relazioni
  - funzioni
  - equazioni
- Tipi
- Concorrenza
- Moduli
- Ambiente e linguaggio adatto alla specifica (+ dichiaratività)
- Ambiente e linguaggio adatto alla programmazione (+ controllo)

LOGIC PROGRAMMING

giustificato

- Ling. di Programmazione
- Ingegneria del Software
- Data Bases
- Architetture di Calcolatori
- Applicazioni avanzate

Progetto Giapponese V^th Generation

VLSI + PROLOG + APPLICAZIONI A.I.

MACCHINA INFERENZIALE

Logic Inference Per Second (LIPS)

La logica sarà:
un ling. macchina particolarmente adatto alla specifica ?
oppure
un ling. di specifica particolarmente efficiente ?

GDS

Pierpaolo Degano Ugo Montanari
Dipartimento di Informatica
PISA

Rappresentare sistemi concorrenti distribuiti

A. Descrivere l'anatomia,
- aspetti temporali (chi viene prima, chi dopo, chi necessariamente assieme, chi può ...)
- aspetti spaziali (a chi è connesso chi, per mezzo di cosa, ...)

B. Descrivere la fisiologia,
- computazione concorrente (in termini di risultare nondeterministiche sequenziali)
- risultato, sia di comp. finite che infinite.

C. Esprimere proprietà:
- terminazione / non terminazione
- stallo
- deadlock
- fairness (weak e strong)

D. Usare e forzare una metodologia (Astesiano's) per descrivere (decompone e compone) sistemi

A + C + D ⟹ Ipergrafi, grammatiche per grafi (produzioni C.F., regole di risultare c.d.

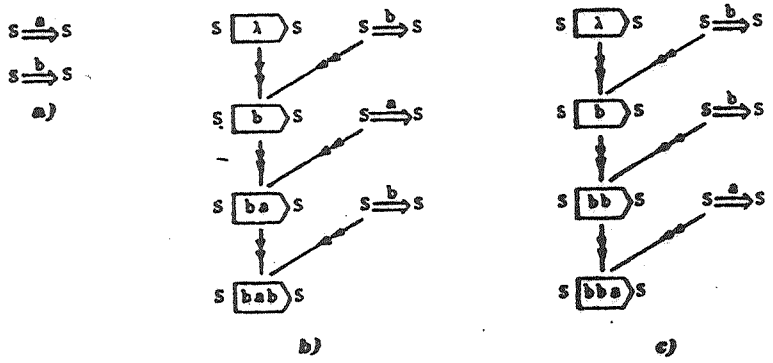
B + C ⟹ Metrica e completamento di spazi metrici

Labelled transition systems (Plotkin, Milner, Hennessy)

1. Uno stato monolitico $S, S_1, \ldots$
2. Regole di transizione (non deterministiche) che provocano azioni atomiche osservabili $(a, b, \ldots)$ nel trasformare uno stato in un altro $S_1 \xrightarrow{a} S_2$
3. Regole di inferenza

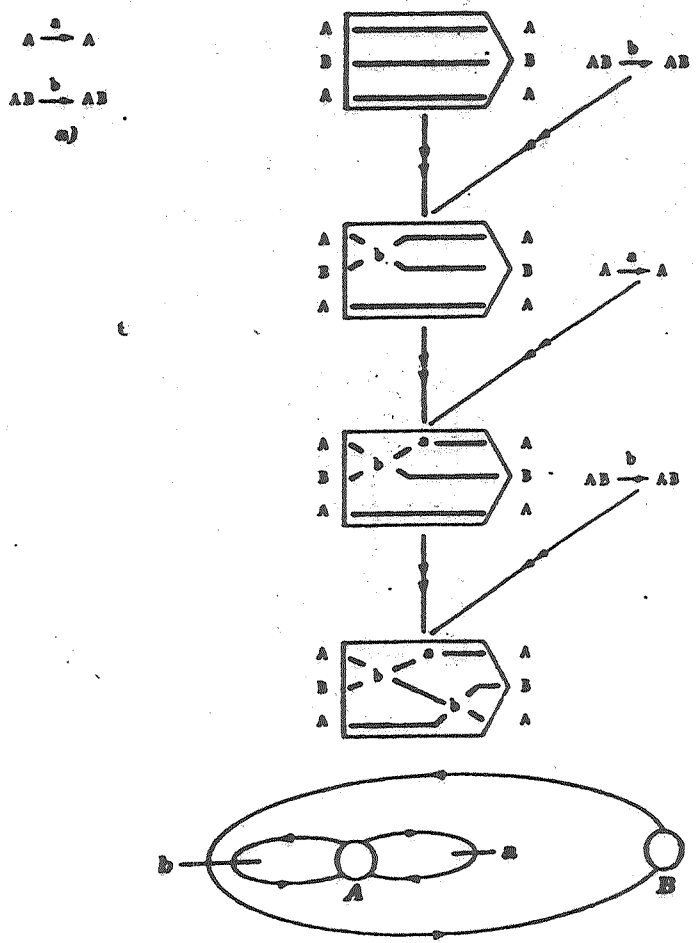
$$S \boxed{\lambda} S \qquad \frac{S_1 \boxed{w} S_2 \;;\; S_2 \xrightarrow{a} S_3}{S_1 \boxed{wa} S_3}$$

4. Computazione = prova
5. Storia della computazione è una stringa (ordinamento totale)



Concurrent t.s.

1. Lo stato è una collezione di stati dei processi A, B, ...
2. una regola di transizione denota la possibile evoluzione di uno o più processi $AB \xrightarrow{b} AB$
3. —
4. —
5. La storia è un ordinamento parziale



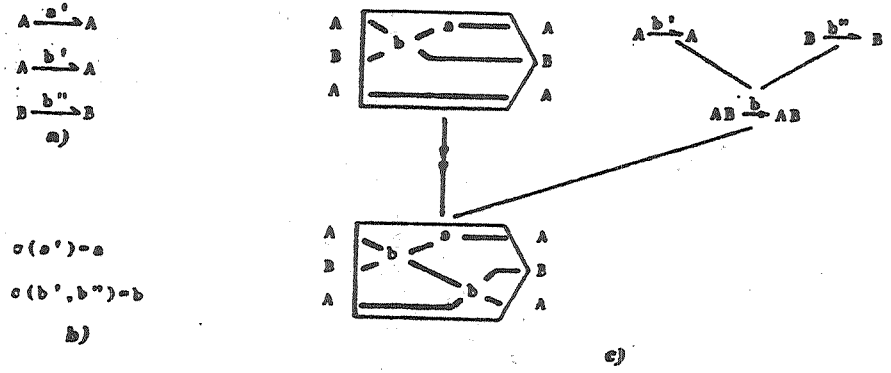
Synchronized concurrent t. s.

- [illegible]

- Vogliamo descrivere l'evoluzione di un singolo processo alla volta e poi trovare un meccanismo per ottenere le regole di riscrittura

2a. Una produzione $(A \xrightarrow{b'} A)$ descrive l'evoluzione di un processo, l'etichetta è chiamata protocollo

2b. Una funzione di sincronizzazione σ consente di combinare produzioni con protocolli compatibili al fine di ottenere una regola di riscrittura etichettata da un'azione osservabile.



confronti nella σ

Milner (synch. trees)

- azioni osservabili = protocolli
- sincronizzazione = comunicazione $\Rightarrow \alpha, \bar{\alpha}$

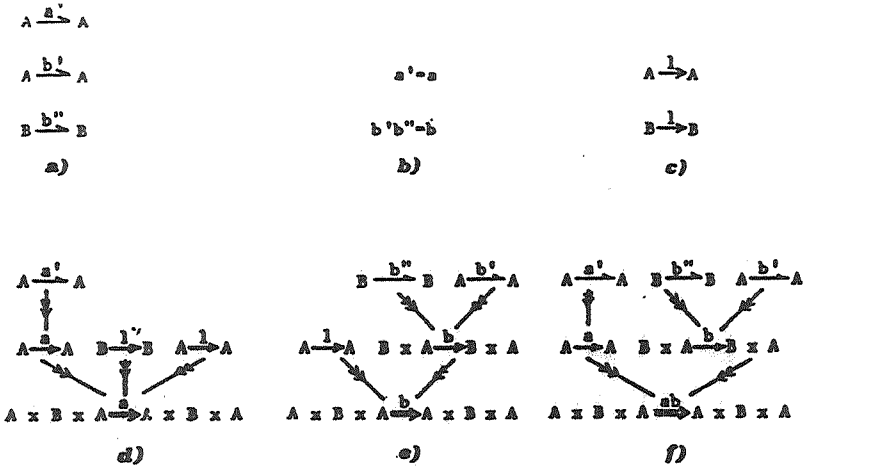
si traduce nel nostro modello con

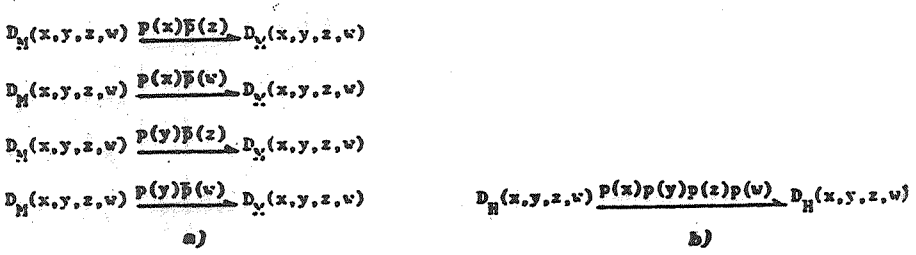
- protocolli = $\{\alpha_i, \bar{\alpha}_i\}$
- azioni osservabili = $\{\tau, \alpha_i, \bar{\alpha}_i\}$
- $\sigma(\alpha_i) = \alpha_i$
  $\sigma(\bar{\alpha}_i) = \bar{\alpha}_i$
  $\sigma(\alpha_i, \bar{\alpha}_i) = \tau$

Hoare (tutti i processi hanno lo stesso protocollo) non si traduce immediatamente a meno di avere una famiglia di funzioni

$\sigma_n(\underbrace{a'_i, \dots, a'_i}_{n}) = a_i$

- $\{a'_i\}$ protocolli
- $\{a_i\}$ azioni





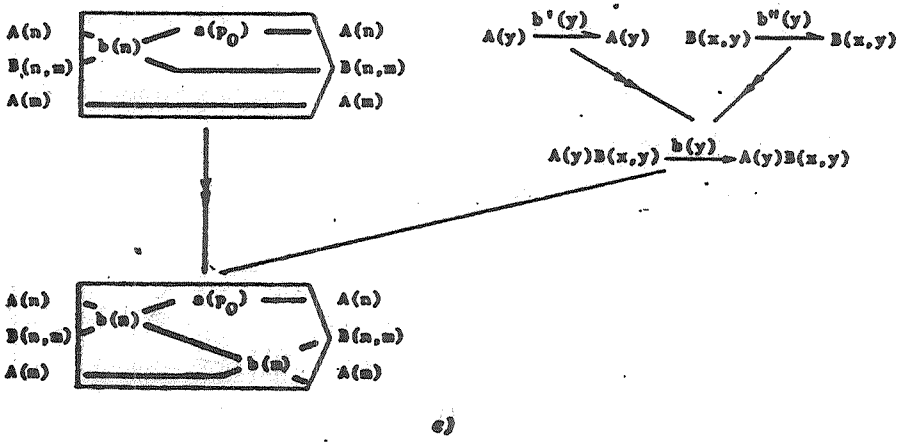
Confronti dei synchronized concurrent t.s. con Milner SCCS

- un insieme I di azioni osservabili
- le espressioni sono ristrette al gruppo (libero commutativo) generato da I

- noi distinguiamo $\rightarrow$, $\Rightarrow$
- abbiamo $\square$ con ordinamento parziale
- non abbiamo $\Rightarrow$

Distributed port-synchronized t.s.

- $\exists$ molti luoghi di sincronizzazione (porte)
- Due processi comunicano alla stessa porta possono comunicare (adiacenti)



Distributed synchronized t.s.

Se ammettiamo che un processo, B(n,m), usando un protocollo simultaneamente su più di una porta, ..., si possono sincronizzare molti, tra., processi.

- Solo due processi comuni ad una porta
  - è il meccanismo più basilare possibile
  - noi = Milner = Hoare
- Omettiamo τ, confondendo protocolli e azioni
- Molte azioni simultanee su canali "sincronizzati" formano una sola etichetta nell'ordinamento parziale, chiamata evento.

$A(x) \xrightarrow{\tau} A(x)$

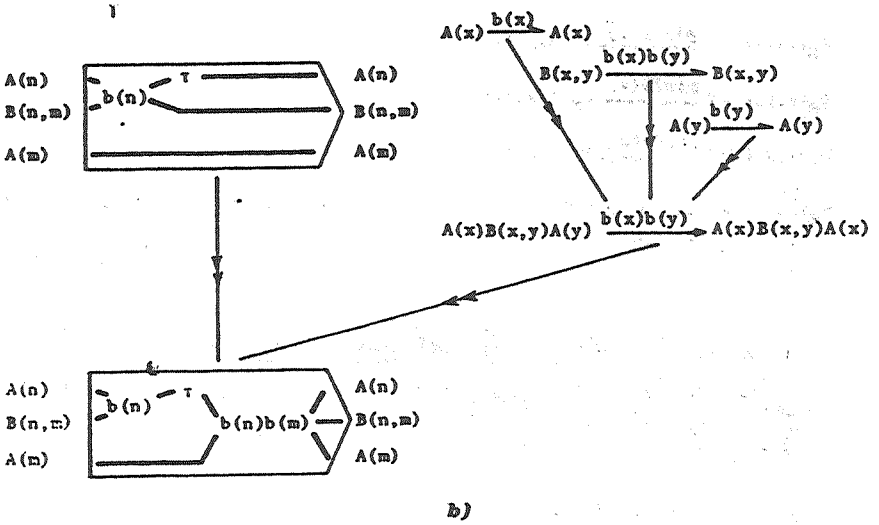
$A(x) \xrightarrow{b(x)} A(x)$

$B(x,y) \xrightarrow{b(x)} B(x,y)$

$B(x,y) \xrightarrow{b(y)} B(x,y)$

$B(x,y) \xrightarrow{b(x)b(y)} B(x,y)$

a)

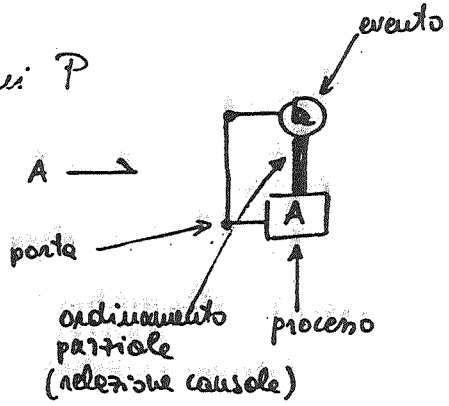


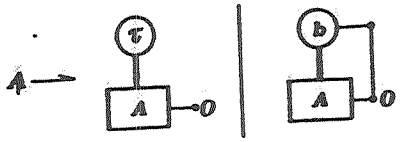
b)

Tail-recursive GDS

- grafo iniziale
- insieme di produzioni P

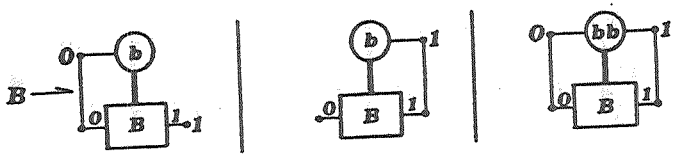
$A(v_1) \xrightarrow{b(v_1)} A(v_1) \longrightarrow A \longrightarrow$

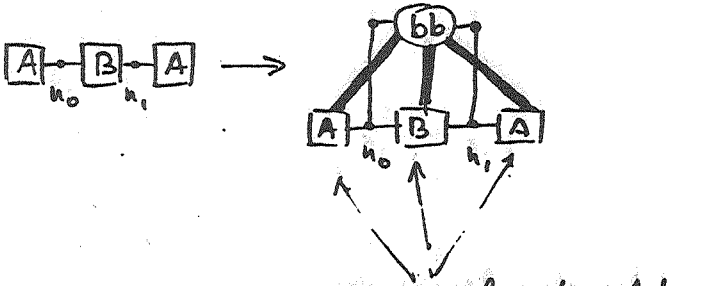




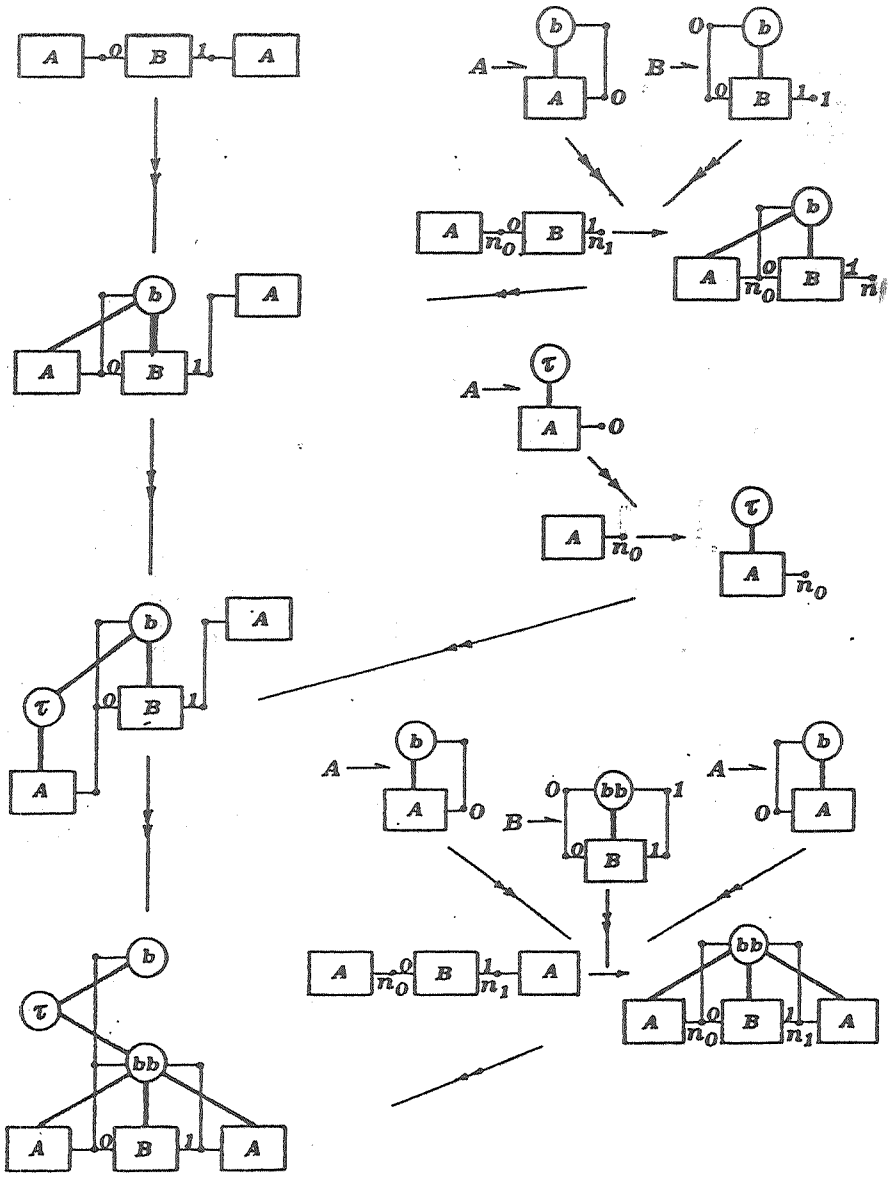
Da P si derivano le regole di riscrittura

$A(v_1)\,B(v_1,v_2)\,A(v_2) \xrightarrow{b(v_1,v_2)} A(v_1)\,B(v_1,v_2)\,A(v_2)$





- le regole di riscrittura vengono applicate (con caso) in maniera standard.



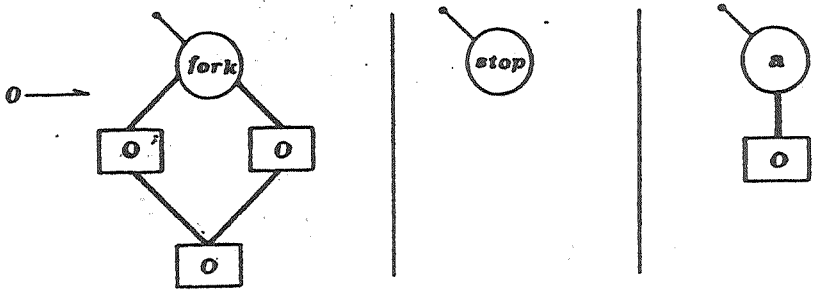
Restrizioni

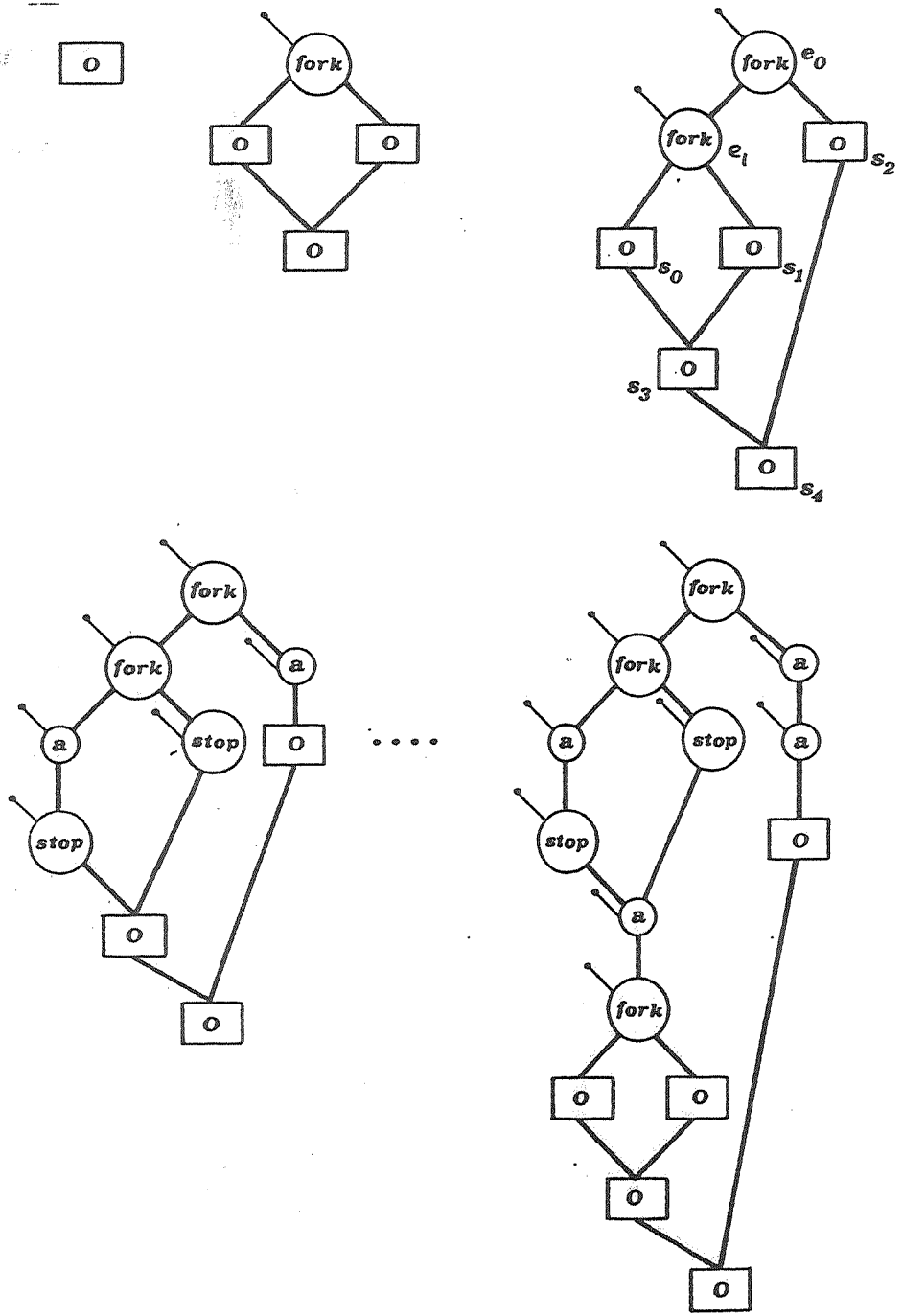
- produzioni (azioni) e regole di riscrittura (contesti - di cui dopo) sono Greibach (irriducibile).
- eventi ≥ processi
- un evento non deve avere adiacente né ad un evento, né ad un processo
- al max due processi adiacenti
- i processi stanno solo "in fondo"

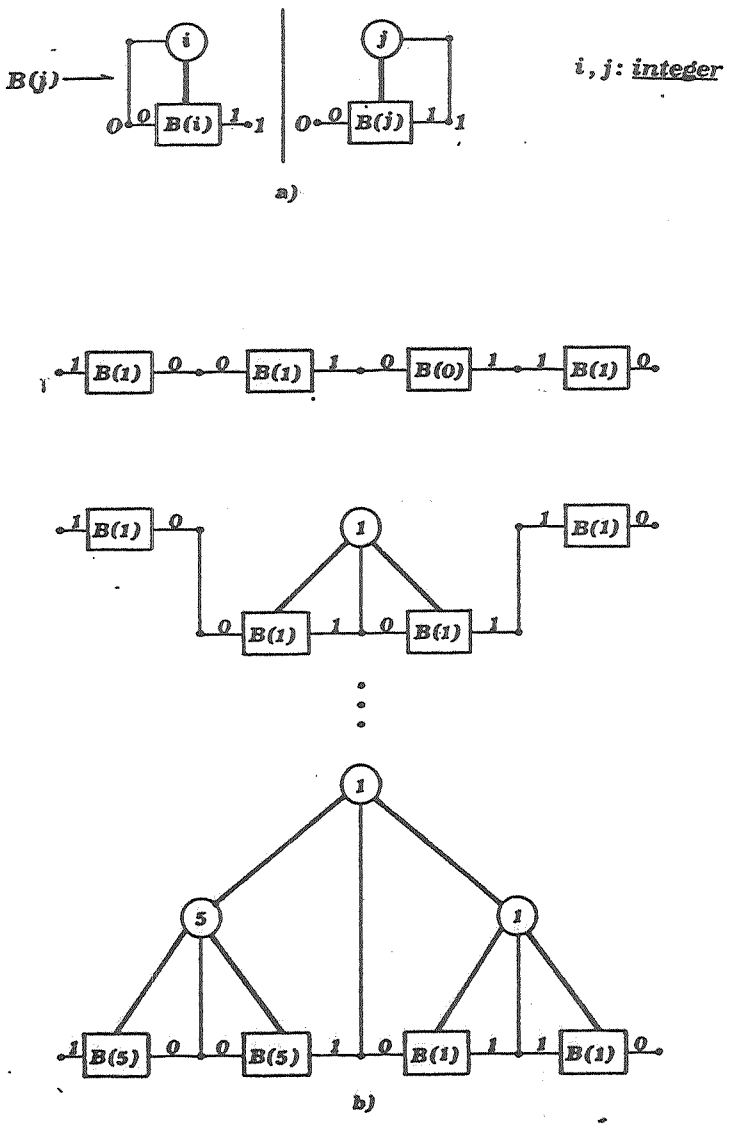
rimuovendo l'ultima si hanno le GDS, in cui abbiamo intera la potenza della ricorsione in quanto un processo può mettere dei processi (mai un evento!)

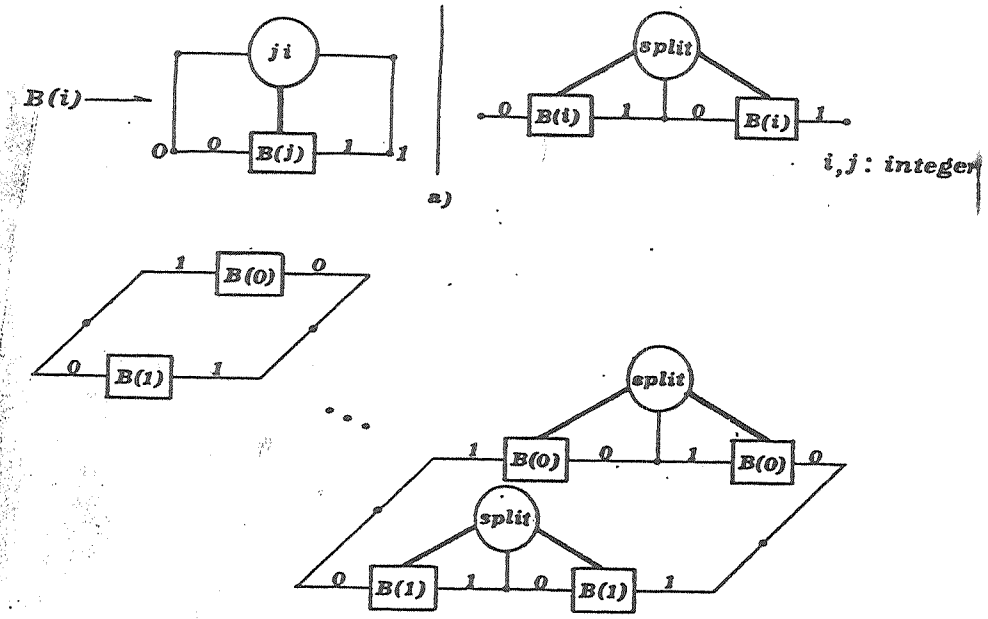
Restrizione

Solo il presente può evolvere




O
fork
O
O
O
fork
e_0
fork
e_1
O
s_2
O
s_0
O
s_1
O
s_3
O
s_4
fork
fork
a
a
stop
O
stop
O
O
fork
a
fork
a
stop
a
stop
O
a
fork
O
O
O
O


B(j)
i
0
0
B(i)
1
1
j
0
0
B(j)
1
1
i, j: integer
a)
1
B(1)
0
0
B(1)
1
0
B(0)
1
1
B(1)
0
1
B(1)
0
1
0
B(1)
1
0
B(1)
1
1
B(1)
0
1
5
1
1
B(5)
0
0
B(5)
1
0
B(1)
1
1
B(1)
0
b)



- Abbiamo una classe di ipergrafi, finiti ed infiniti, $\mathcal{D}$.
- La seguente è una (ultra-) distanza

$$d(D_1, D_2) = \begin{cases} 2^{-\max\{n \mid [D_1]_n = [D_2]_n\}} & \text{se c'è} \\ 0 & \text{altrimenti} \end{cases}$$

$(\mathcal{D}, d)$ è completo

$Fin(\mathcal{D})$ è denso in $\mathcal{D}$

$(\mathcal{D}, d)$ è il completamento di $(Fin(\mathcal{D}), d)$

- Data una E.D.S. è una computazione $\{D_i\} = (D_0, D_1, \dots)$ è una successione di ipergrafi tali che $D_i \xrightarrow{} D_{i+1}$, $i \geq 0$.
- Una computazione induce un ordinamento totale sugli eventi, generation ordering.

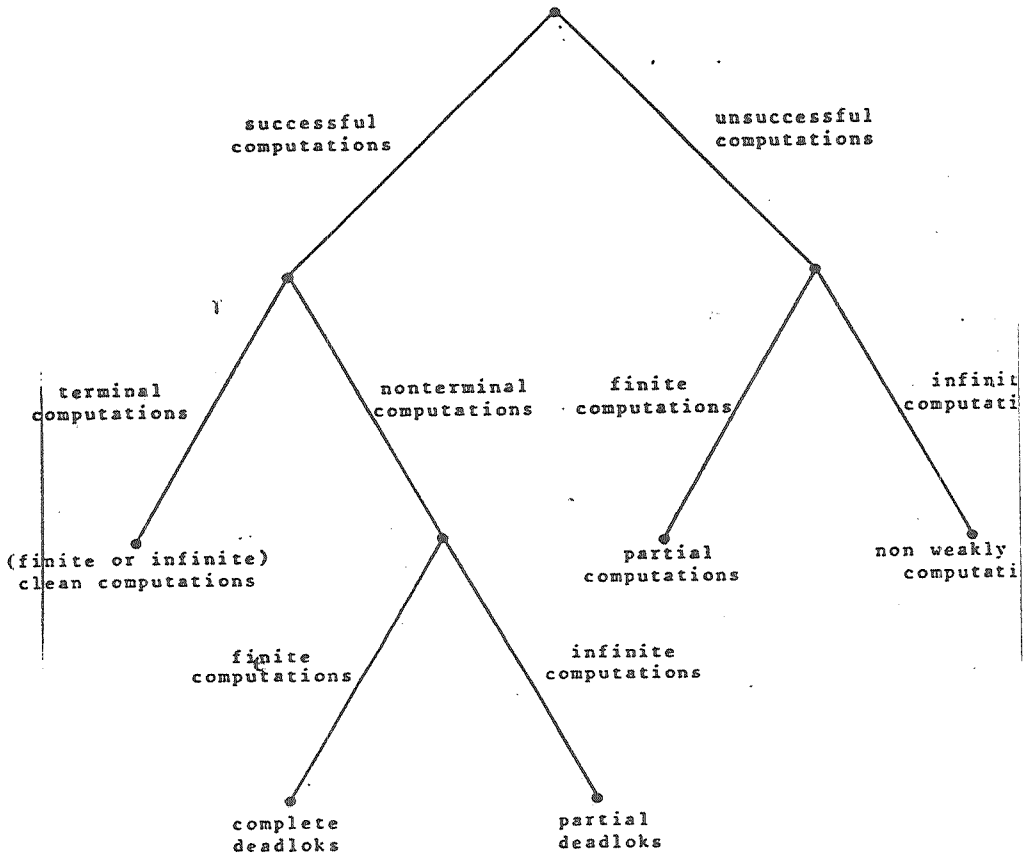
> Ogni computazione infinita è una successione di Cauchy

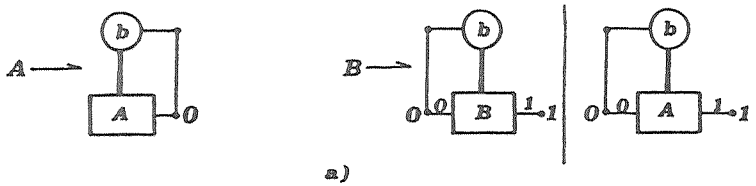
- Il risultato di una computazione
  - finita: è il suo ultimo ipergrafo
  - infinita: è l'ipergrafo infinito limite della corrispondente successione di Cauchy.

La nozione di computazione equivalente viene data dal seguente teorema

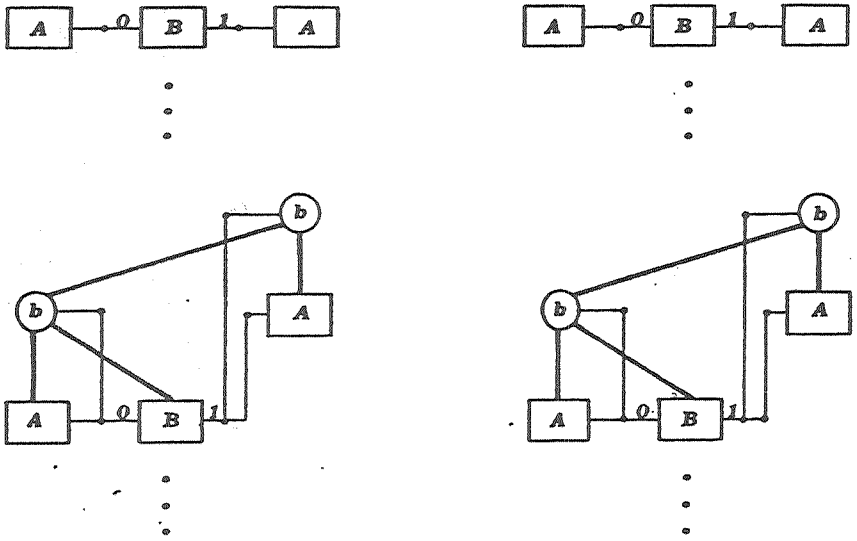
> Ogni generation ordering è compatibile con la relazione di causalità (ordinamento parziale) presente nel risultato.

Ci sono vari tipi di computazione ...



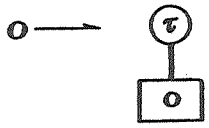

A
b
A
0
B
b
0
0
B
1
1
b
0
0
A
1
1

a)

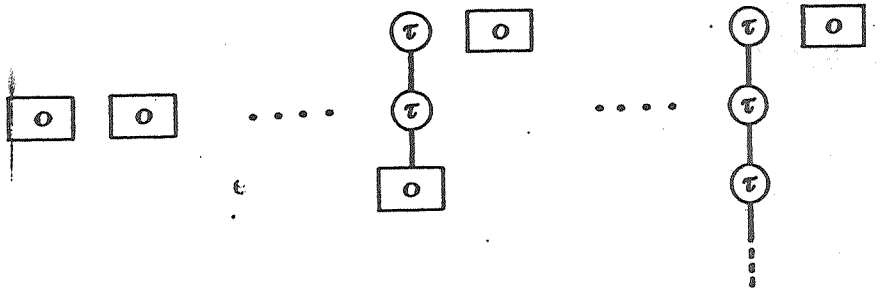

A
0
B
1
A
b
A
0
B
1
b
A
b
b
b
b
b
b
b
b
b
A

b)

c)


O
τ
O

a)

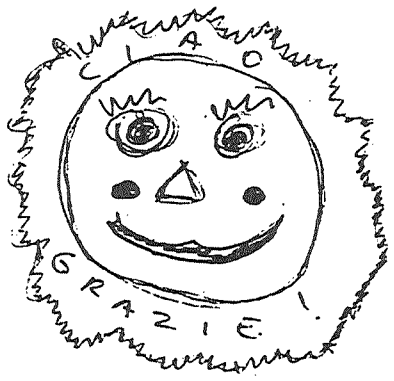

O
O
τ
O
τ
O
τ
O
τ
O

- Il linguaggio generato da una GDS è l'insieme delle sue computazioni successful

comp. weakly fair = comp. successful

È una scheduler che genera tutte e sole le computazioni successful

- Il linguaggio generato da una GDS non è mai vuoto
- Ogni computazione finita è prefisso di una computazione successful.



- Semantica ad "insieme" — inadeguata
- Semantica ad "albero" (LTS, syuc. trees) per evidenziare i punti di scelta
- Due computazioni che differiscono solo per il generation ordering (2 eventi concorrenti "scambiati") vanno identificate

⇓

Primo passo verso la definizione di una observational equivalence

- Quando un tipo di evento viene reso non osservabile, come si raffina l'observational equivalence?
- Come si definiscono meccanismi di astrazione?
- Come si compongono "moduli" e quali è la semantica?

⋮