88 informatica 2

YU ISSN 0350-5596

Iskra Delta

Iskra Delta

Iskra Delta

Iskra Delta

Iskra Delta

Iskra Delta

Iskra Delta

Iskra Delta

Iskra Delta

Iskra Delta

Iskra Delta

Iskra Delta

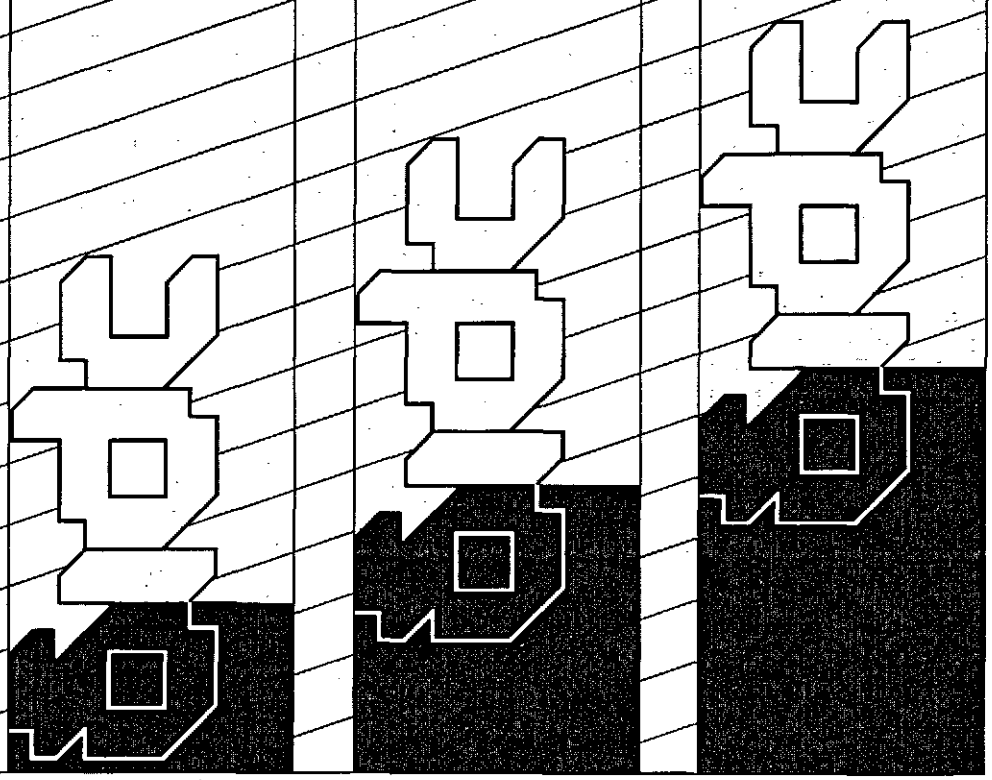
Iskra Delta

Iskra Delta

Iskra Delta

Iskra Delta

proizvodnja računalniških sistemov in inženiring, p.o.
61000 Ljubljana, Parmova 41
telefon: (061) 312-988
telex: 31366 YU DELTA



informatica

ČASOPIS ZA TEHNOLOGIJO RAČUNALNIŠTVA
IN PROBLEME INFORMATIKE
ČASOPIS ZA RAČUNARSKU TEHNOLOGIJU I
PROBLEME INFORMATIKE
SPISANIE ZA TEHNOLOGIJA NA SMETANJETO
I PROBLEMI OD OBLASTA NA INFORMATIKATA

YU ISSN 0350-5596

LETNIK 12, 1988 – ŠT. 2

Casopis izdaja Slovensko društvo Informatika, 61000 Ljubljana, Parmova 41, Jugoslavija

Uredništvo in uprava:

Casopis Informatika, Iskra Delta, Stegne 15B, 61000 Ljubljana, telefon (061) 574 554; teleks 31366 YU Delta.

Letna naročnina za delovne organizacije znaša 24000 din, za zasebne naročnike 6000 din, za študente 2000 din; posamezna številka 8000 din.

Številka žiro računa: 50101-678-51841

Pri financiranju časopisa sodeluje Raziskovalna skupnost Slovenije

Na podlagi mnenja Republiškega komiteja za informiranje št. 23-85, z dne 29. 1. 1986, je časopis oproščen temeljnega davka od prometa proizvodov.

Printed by: Tiskarna Kresija, Ljubljana, in March, 1988

V S R B I N A

informatica

Published by Informatika, Slovene Society for Informatics, Parmova 41, 61000 Ljubljana, Yugoslavia

Headquarters:

Informatica, Iskra Delta, Stegne 15B, 61000 Ljubljana, Yugoslavia. Phone: 61 57 45 54. Telex: 31366 yu delta

Annual Subscription Rate: US$ 30 for companies, and US$ 15 for individuals

Tisk: Tiskarna Kresija, Ljubljana, v marcu 1988

JOURNAL OF COMPUTING AND INFORMATICS

YU ISSN 0350-5596

VOLUME 12, 1988 – No. 2

CONTENTS

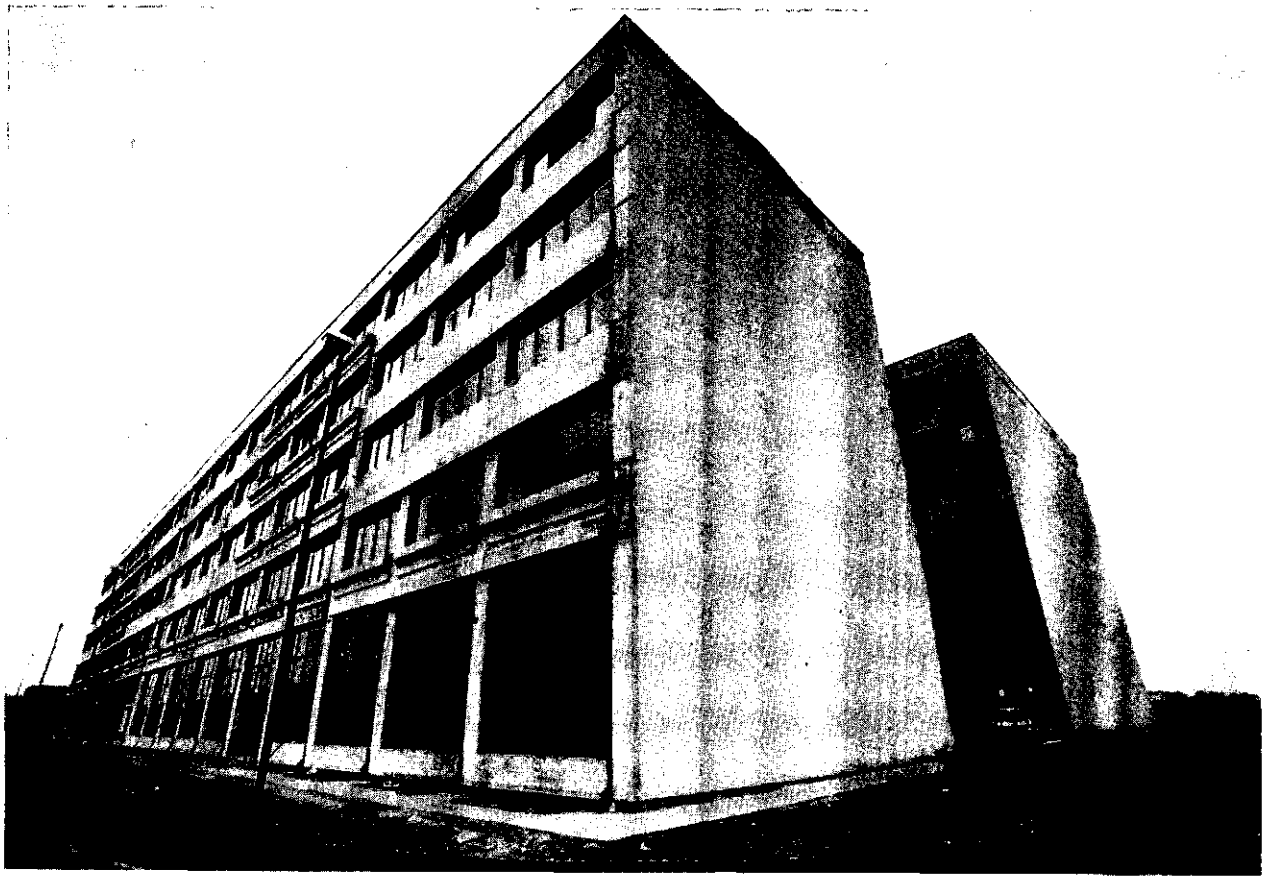
UDK 681.3.001.2

Anton P. Železnikar
Iskra Delta

Ta članek je avtorjev govor ob otvoritvi Deltinega razvojno-proizvodnega centra v Iskrini tehnološki dolini v Stegnah pri Ljubljani. Članek poudarja več kot tridesetletno zamisel avtentične strukture in organizacije računalniške tovarne, ki ni le privesek neke raznorodne, računalništvu tuje industrije, temveč je sodobna proizvodna zmogljivost, ki obsega dovolj širok in tehnološko zahteven računalniški proizvodni program. Ta proizvodnja naj bi bila podprta s sodobno zamišljenimi raziskavami in razvojem računalniških in informacijskih sistemov. Že danes je Iskra Delta povezana z raziskovalnimi, razvojnimi in proizvodnimi središči doma in na tujem. V prihodnje naj bi bila nacionalna računalniška industrija tudi deležna posebne skrbi in podpore vlade, industrije in akademije. Iskra Delta sproža tudi že novo industrijsko raziskovalno pobudo za sodobne tehnološke raziskave na področju računalniške znanosti, tehnologije in uporabe. Pod okriljem Slovenskega društva Informatika je bil ustanovljen t.i. Forum informationis, ki ga sestavljajo najvidnejši računalniški strokovnjaki in novinarji, z namenom, da bi lahko nudil javno in kritično podporo hitro rastoči domači računalniški industriji. Na koncu članka je sporočen poziv vodstvu in inženirjem Iskre Delte, ki naj bi pospešil intelektualno motivacijo in ustrezno poklicno reagiranje inženirjev.

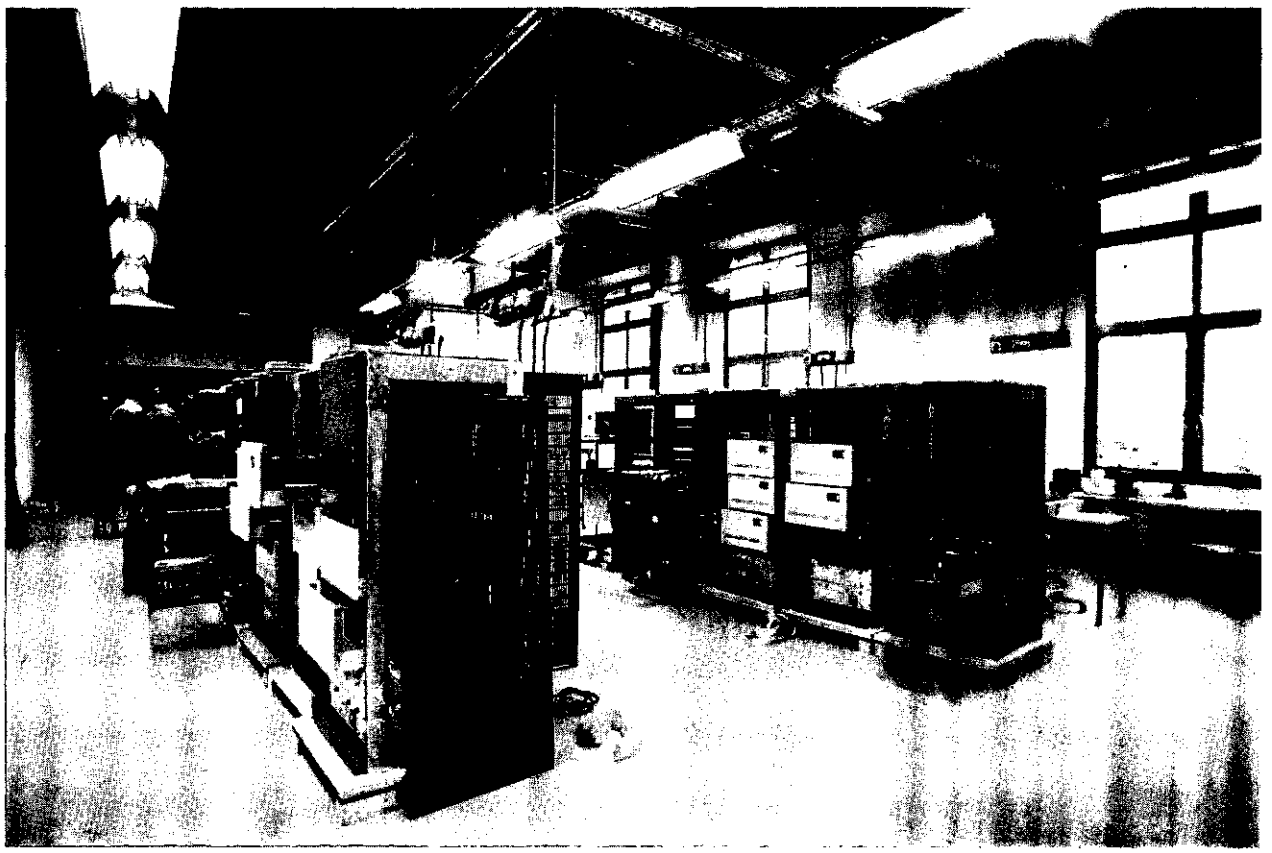
On the Developmental Iniciative of the Computer Industry.

This paper presents the opening speech, held by the author, at the occasion of the opening ceremony of the *Iskra Delta's Research, Development, and Manufacturing Center* at the so-called Iskra Technology Valley in Stegne, near Ljubljana. The paper apostrophizes more than the thirty-years growing concept of authentic structural and organizational principles of a computer factory, which is not only an appendix of a heterogenous, to the field of computing strange industry, but is a modern manufacturing facility embracing sufficiently broad and technologically pretentious computer manufacturing program. This manufacturing should be supported by contemporarily imagined research and development of computer and information systems. Already today, Iskra Delta is well-connected with research, development and manufacturing places in the country and abroad. In future, the national computer industry should receive the necessary *care and support* from governmental authorities, industry and academia. Practically, Iskra Delta is also giving the so-called *industrial research initiative* for modern technological and conceptional research in the area of computer science, computer technology, and to other computer-related application. In the framework of the Slovene Society for Informatics the so-called *Forum Informationis,* uniting the most distinguished and experienced computer professionals and journalists was established, to offer the public and critical support to the fast growing domestic computer industry. At the end of the article, a call to the *managerial* and professionally diverse *engineering staff* of Iskra Delta is communicated, to rise the necessary intellectual motivation and the corresponding professional reaction.



Slika 1 (zgoraj). Na sliki je prikazan izgled proizvodno-razvojnega objekta Iskre Delte med izgradnjo. Sprednji del objekta (levo) je v glavnem namenjen raziskovalno-razvojnemu delu, zadnji del (desno) pa pretežno proizvodnemu delu. V kletnih etažah se nahajajo skladišča, na vrhu zadaj pa infrastrukturne enote (kuhinja, menza, ambulanta, predavalnica).

Slika 2 (spodaj). Ta slika kaže proizvodno montažo in preizkušanje večprocesorskih miniračunalniških sistemov Gemini in Delta 4860, ko se njihova izdelava že približuje končni fazi. To so znani in že uveljavljeni računalniški sistemi Iskre Delte iz razreda super miniračunalnikov, z lastnimi modulskimi aparaturnimi in prograskimi komponentami.



Slika 3 (desno). Pri izdelavi modulov na tiskanih vezjih, ki je sicer avtomatizirana, so večkrat potrebne tudi dodatne ali korekturne ročne operacije. Tako se moduli dopolnjujejo oziroma popravljajo. Vse to pa zahteva vestno oziroma natančno delo.

Slika 4 (spodaj). Tiskano vezje z vstavljenimi in električno povezanimi elementi se preizkuša z uporabo posebne preizkuševalne naprave, ki opravlja meritve in s tem preverja avtomatično ustreznost vezja. To je visoko učinkovito diagnosticiranje modula na tiskanem vezju, torej ugotavljanje kritičnih odstopov.

Slika 5 (zgoraj). Na sliki je prikazana proizvodnja mikroračunalniških sistemov Triglav (stolpna ali obmizna izvedba) s 16-bitnimi procesorji J11, M68010 in I80286. Te konfiguracije je mogoče raznovrstno dopolnjevati z disketnimi, diskovnimi in tračnimi enotami.

Slika 6 (desno). Namizna izvedba mikroračunalniških sistemov Triglav je namenjena njihovi uporabi s funkcijo visoko zmogljivih delovnih postaj za različne namene (CAD/CAM, grafika za različne uporabne naloge, komunikacijske mreže itd.). Sistemi Triglav uporabljajo lastne, tj. domače module s tremi različnimi mikroprocesorji na vodilu VME. To vodilo je industrijski standard. Tudi operacijski sistemi za računalnike družine Triglava so različni in omogočajo uporabo najširšega spektra uporabniških programov. Ti operacijski sistemi so Delta-M, Xenix, Unix in MS-DOS.

Some of the riskiest work we do is concerned with altering organization structures. Emotions run wild and almost everyone feels threatened. Why should they be? The answer is that if companies do not have strong notions of themselves, as reflected in their values, stories, myths, and legends, people's only security comes from where they live on the organization chart. Threaten that, and in the absence of some grander corporate purpose, you have threatened the closest thing they have to meaning in their business lives.

T.J. Peters and R.H. Waterman: In Search of Exellence, p. 77 (Harper & Row, 1981)

Zamisel, ki smo jo jugoslovanski računalniški inženirji nosili več kot trideset let[2] v svojem hotenju in znanju, se v teh dneh materialno uresničuje. Pred nami je prava in na lastnih tehnoloških in razvojnih temeljih, na dolgotrajni avtentični zamisli[3] zgrajena in organizirana tovarna računalnikov, ki ni le privesek neke drugorodne, računalništvu mačehovske ali *tuje* industrije, temveč je sodobna proizvodna zmogljivost, ki obsega dovolj širok in tehnološko zahteven računalniški proizvodni program. Ta nova zmogljivost naj bi se podpirala tudi s sodobno koncipiranimi raziskavami in razvojem, ki naj bi zagotavljale novonastajajoče računalniške produkte, tj. informacijske in računalniške sisteme za *svetovno konkurenčni plasma in uporabo.* To je vizija in odločenost ob tej otvoritvi in prav zaradi tega želim kritično poudariti, da bosta svetovno konkurenčni plasma in uporaba mogoča le tedaj, če bo sodobno organizirana znanstveno-produkcijska veriga dobivala ustrezne *tržno-strateške inpute* in le če se bo vzpenjala na *skrajne meje znanstvene in tehnološke inovativnosti* ob izredno samokritičnem uresničevanju vseh dejavnosti sklenjene verige raziskav, razvoja, proizvodnje in trženja.

Vsestranska znanstveno-tehnološka povezanost Iskre Delte s svetovnimi in domačimi raziskovalnimi, razvojnimi, tehnološkimi, proizvodnimi in poslovnimi središči bo odločilna in mora neglede na trenutne krizne okoliščine slej ko prej ostajati, nastajati in se razvijati kot glavna strateška usmeritev računalniškega podjetja. Takšna usmeritev, ki edina zmore zagotavljati obstoj in prepotrebne razvojne impulze lastne in druge sodobne industrializacije na naših tleh in v evropskih koordinatah, bo lahko nošena le s strokovno in intelektualno vrhunsko usposobljenimi kadri v vodenju, raziskavah, razvoju, proizvodnji in trženju. V naporih za svoje preživetje in razvoj se bo Iskra Delta vedno znova soočala s problemom zahtevne *kadrovske kvalifikacije,* izkušenosti in intelektualne prodornosti. Značilni kadrovski kompromisi današnje stagnantnosti, neaktivnosti in disfunkcije ne bodo zadostovali niti za golo preživetje.

Računalniška industrija in razvoj te industrije sleherne države sta danes pomemben integracijski dejavnik na področju narodnega gospodarstva, raziskav, razvoja, proizvodnje in trženja. Ta industrija je zato *skrbno opazovana in varovana z državno regulativo,* saj je izhodiščna za razrast druge industrije, malih podjetij materialne in programske proizvodnje, trgovine, nacionalnih šolskih, zdravstvenih in drugih infrastrukturnih dejavnosti. Računalniška industrija je temeljna podlaga prihajajoče informacijske epohe. Narodi ali mednarodne skupnosti, ki še domišljajo in gradijo svoje vizije preživetja, morajo jasno in načrtovano predvidevati svoje vire informacijskega znanja in tehnologije.

Tudi funkcija današnje Iskre Delte postaja čedalje bolj raznovrstno *integrativna* na področju poslovnega povezovanja, postaja pa tudi bistveno *inicializirajoča* skladno in spontano s tistimi potrebami sodobnega tehnološkega in socialnega razvoja, ki zmore odločilno posegati v prestrukturiranje papirnatih vrhunskih raziskav in tudi v navidezno oziroma nebistveno akademsko tehnološko napredovanje. *Raziskovalna industrijska iniciativa* Iskre Delte že oblikuje tisto bistveno *polje razvojne relevantnosti,* ki bo lahko podlaga ključnim preobratom v razvojni strategiji sozda Iskre in Gorenja in t.i. raziskovalnih skupnosti. Iskra Delta vendarle in to navkljub občutjem krizne brezizhodnosti odpira obetajoče in motivacijsko bistvene

---

[1] Slavnostni govor ob *otvoritvi proizvodno-razvojnega središča* Iskre Delte v Stegnah, dne 4. decembra 1987.

---

[2] Zamisli o oblikovanju *slovenske računalniške industrije* so nastajale že v drugi polovici 50-ih let. Na Institutu Jožefa Stefana so potekala interdisciplinarna predavanja, s katerimi se je de facto pripravljal tudi razvoj računalnika, ki naj bi ga proizvajala domača industrija. V 60-ih letih je ta zamisel ugašala, saj se je na Zavodu za avtomatizacijo pojavila licenčna montaža (neke vrste proizvodnja) računalnika Zuse Z23.

---

[3] Avtentična zamisel računalniške tovarne pomeni med drugim na lastnih možnostih in izkušnjah strukturirano in organizirano proizvodnjo, podprto s sodobno (lastno) računalniško avtomatizacijo tipa CIM (Computer Integrated Manufacturing) in tipa CIC (Computer Integrated Communication), kot jo uvajajo sodobno organizirane tovarne v avtomobilski (General Motors) in računalniški industriji (npr. avtomatizacija proizvodnje mikroračunalniških sistemov tipa PS in velikih računalniških sistemov pri IBM).

Slika 7 (zgoraj). Na tej sliki je viden del laboratorija za razvoj mikroračunalniškega sistema Triglav, in sicer za modul z 32-bitnim mikroprocesorjem Intel 80386. Iz slike je razvidno, kako so svetli laboratorijski prostori bogato opremljeni z najrazličnejšimi aparaturnimi pripomočki za kompleksen razvoj (načrtovanje, risanje, simulacija itd.) vezij in podsistemov. Preko posebnih terminalov je laboratorij povezan z visoko zmogljivimi miniračunalniškimi sistemi lastne proizvodnje (glej npr. sliko spodaj). Ti razvojni računalniki so locirani na več mestih v proizvodno-raziskovalnem središču Iskre Delte.

Slika 8 (spodaj). Računalniški center projektov za operacijske sisteme in super miniračunalnike je le eden od petih računalniških centrov v proizvodno-razvojnem središču Iskre Delte v Stegnah. V ozadju je v sredini slike viden večprocesorski računalniški sistem ID Gemini, s skupnim diskovnim obsegom 4,2 Gbyte. Ta sistem je namenjen med drugim tudi razvoju IDA orodij in lastnih operacijskih sistemov iz družine Deltix. Na levi strani slike je viden sistem VAX 11/780, ki se uporablja pri razvoju super miniračunalnikov (za simulacijo, načrtovanje in druge razvojne naloge).

razvojne perspektive, ki jih letargično in nemotivirano okolje že sprejema s tihim odobravanjem, saj vidi in čuti v njih svoj izhod, razvojno upanje in prestrukturirno zaupanje. V tem spoznavnem kontekstu je razvojna uspešnost Iskre Delte izredno pomembna za občutje industrijskih in drugih infrastrukturnih razvojnih možnosti jugoslovanskih nacionalnih in federalnih znanstveno-tehnoloških strategij.

V bližnjem razdobju se bo z nadaljnim Deltinem razvojem pojavila tudi v obeh sozdih pa tudi v strokovni in širši javnosti bistveno nova, oživljajoča pobuda. Slovensko društvo Informatika bo oblikovalo t.i. *Forum informationis*[4], ki ga bodo sestavljali najvidnejši računalniški strokovnjaki in novinarji, kot kritično, problemsko in intelektualno usmerjeno telo za področje računalništva in informatike. Delovno polje relevantnosti tega telesa bo raznovrstna *strokovna, raziskovalna, organizacijska* in *proizvodna pobuda*. Hkrati bo Slovensko društvo Informatika reformiralo svoj strokovni časopis *Informatica* tako, da bo njegova vsebina bistveno povezana z raziskovalno in razvojno problematiko obeh sozdov, da se bo skozi časopis razvijala tudi potrebna motivacija za raziskovalnost, razvojnost, strokovnost in znanstveno-tehnološko obveščenost raziskovalcev in inženirjev na področju računalništva in informatike. Z odpiranjem Deltine tovarne se tako de facto začenja tudi bistveno nova, oživljajoča pobuda strokovne aktivnosti kot nujnost in kot predhodnica in posledica industrializacije, kot oko in uho potrebne razvojne relevantnosti.

Z vstopom Iskre Delte v računalniško industrializacijo se odpirajo nove možnosti za informacijsko-značilno in spremljajočo drugo industrijo v ožjem in širšem okolju. Državni, gospodarski in politični centri odločanja naj bi take in drugačne, smiselne in integracijske pobude in tokove, ki so povezani z informatizacijo ključnih dejavnosti in z njihovim nujnim razvojem, z razumevanjem podpirali. Iskra Delta je eden tistih *inovativnih gospodarskih poganjkov*, ki je še sposoben izvenkrizne rasti, motivacije in pozitivnih gospodarskih učinkov. Zaradi tega pričakujem, da bo politična moč podpirala prav svoj lasten, demokratičen razvoj v največji meri tudi tako, da ne bo postavljala umetnih zaprek Deltinemu razvoju tam, kjer bi ga bilo smiselno *kvečjemu* pospeševati. Seveda pa mora tudi Iskra Delta v prihodnje sprejeti *strokovno* in vsestransko odgovornost za skladen in ekonomsko učinkovit razvoj računalništva in informatike na naših tleh in v širšem evropskem in svetovnem prostoru.

Podobno kot se s tem govorom obračam na širšo slovensko in jugoslovansko javnost, se obračam še posebej na Deltine inženirje. Predvsem od vas Deltini inženirji, od vaše strokovne zavezanosti, organiziranosti, brezkompromisarskega izražanja vsakršnjih kvalitativnih zahtev, od vašega hotenja in raziskovalno-tehnološkega izpopolnjevanja je in bo odvisen razvojni in gospodarski potencial in uspešnost Iskre Delte. Slej ko prej se bo potrebno odpovedati *konzervativni in že danes razvojno neperspektivni tehnologiji*[5] in osvajati naprednejšo, za vse nas napornejšo in zahtevnejšo tehnologijo in znanje. Znani koncepti kopiranja in golega posnemanja ne bodo več zadostovali. Zato pričakujem, da bo vaša *strokovna in organizacijska glasnost* glasnejša, strokovno bolj intelektualizirana in konceptualno smelejša, kot je bila doslej; da inženirji ne boste le nemočni opazovalci in nekritični sprejemniki tistih usmeritev, katerim se argumentirano potihoma ali vsaj ne dovolj glasno upirate. Inženirji -

---

[4] Forum informationis je latinskega izvora in pomeni trg (javno mesto) predočbe (informacije). Pomen foruma je dvojen: informacijsko strokoven, tj. utemeljen s strokovnostjo informacije kot informacije (računalnik je danes pojem za informacijski stroj) in javen, tj. kvalificiran za javno razpravo o računalništvu in informatiki.

---

[5] *Konzervativna tehnologija* je v svojem bistvu subkulturno oblikovana (funkcionalno in oblikovno "dizajnirana"), modificirana, največkrat reducirana tehnologija, ki ne sledi več glavnim, tudi konjunkturnim razvojnim, tržnim in standarizacij-skim trendom, se pa na značilen nesposobnosten način odeva v svoje zaščitne znake in ščiti tako le sama sebe pred nikomer. Tehnološka konzervativnost je lahko le trenutna tržna poteza, ki jo je potrebno čimprej nadomestiti s tržno relevantno noviteto.

---

[6] Po otvoritvenem govoru[1] so se pojavili očitki, da je ta govor *poziv Deltinim inženirjem na upor*. Moj odgovor na te očitke je bil in ostaja, da se Deltini inženirji dejansko morajo argumentirano in zavestno, torej sistematično in strokovno organizirano, z vso intelektualno silo, ki je vest in zavest sodobnega inženirja, upirati kriznemu napredovanju, ki grozi z razrušitvijo doseženega industrijskega, predvsem pa v tem kompleksu nujnega, vse drugo pogojujočega raziskovalno-razvojnega potenciala in s tem k razrušitvi določenega preživetvenega minimuma. Inženirji kot vlečni konji so moralno poklicani, da vlečejo skupaj z drugimi voz visokotehnološkega podjetja iz krize, da gradijo svojo moralno in strokovno motivacijo predvsem tudi na elementu vleke iz krize. Brez te vleke, ki je moralna in intelektualna, je tudi sama populacija inženirjev obsojena na propadanje, na drsenje v letargijo in v skrajni posledici na izumrtje.

Slika 9. V novih prostorih proizvodno-razvojnega centra Iskre Delte v Ljubljani, v Stegnah 15b, je dobilo svoj prostor tudi uredništvo časopisa Informatica. Iz slike je razvidna funkcionalnost prostora, ki je razen z uredniškimi mizami in policami opremljen tudi s terminalom, mikroračunalnikom in pomožnim tiskalnikom. To je hkrati tudi delovni prostor glavnega urednika.

Ob vsem tem je gotovo treba poudariti, da je Iskra Delta materialno in vsebinsko z vsestranskim razumevanjem podpirala delo časopisa Informatica od leta 1980 naprej. Le pod takimi pogoji je bilo mogoče vzdrževati utečeno, standarizirano in zanesljivo urejanje in izhajanje časopisa Informatica. Seveda je pri tem nujno poudariti tudi urejene razmere v Raziskovalni skupnosti Slovenije, ki je z vsakoletnim financiranjem omogočala praktično neproblematično izhajanje časopisa.

Okolje uredništva časopisa Informatica je izjemno primerno, saj je uredništvo locirano v samem središču t.i. raziskovalno-razvojne enote Iskre Delte; ta enota predstavlja bržkone jedro prihodnjega (korporativnega, tj. Iskrinega) instituta za računalništvo (in informatiko). Prav s tem pa se vpliv stroke in aktivne publicistične kulture v taki in drugačni obliki prenaša tudi na delo vrste raziskovalcev, razvojnikov in inženirjev Iskre Delte. Uredništvo lahko na ta način tudi sistematično vzpodbuja in povezuje potencialne avtorje v Iskri Delti in izven nje. V nove prostore uredništva prihajajo tudi avtorji in študentje iz drugih krajev, iz zunanjih institutov in jugoslovanskih univerz. Na ta način se smiselno oblikujejo interesne povezave naših in tujih avtorjev in strokovnjakov.

(Nadaljevanje podnapisa k sliki 9.) Zanimivo je tudi, da krizne razmere ne vplivajo negativno na rast obsega, kakovosti in na distribucijo časopisa Informatica. Število prispevkov narašča in urejanje se ne sooča več s problemom premjahnega obsega, katerega spodnja meja je bila ob ustanovitvi časopisa v letu 1977 postavljena na 72 strani za eno številko. Čedalje večja je tudi pripravljenost avtorjev, da pišejo svoje prispevke v angleškem jeziku, s čemer se bistveno povečuje komunikativnost časopisa oziroma prispevkov posameznih avtorjev s tujino.

V bližnji prihodnosti bo časopis Informatica nekoliko spremenjen. Načrtuje se sodobnejša oblika oziroma vsebina platnic. Z razvojem t.i. namiznega urejanja in izdajanja pa bi bilo mogoče vpeljati tudi izpisovanje vseh tekstov z laserskim tiskalnikom. V tem primeru bi morali avtorji svoje članke dostavljati v uredništvo (še) na disketah oziroma z elektronsko pošto. Na ta način bi dobili časopis, ki bi bil v vseh ozirih (in po možnosti ob nebistveno spremenjenih stroških) tudi tehnološko kvalitetno oblikovan. Ob vsem tem je seveda potrebna tudi spremenjena uredniška politika, ki bi zagotavljala kvalitetni dotok prispevkov in morda tudi pogostnejše izhajanje časopisa (v prvi fazi bi bilo mogoče brez posebnih naporov izdajati letno šest namesto dosedanjih štirih številk v enakem obsegu).

Na koncu tega zapisa velja omeniti, da je potrebno navajati novi naslov uredništva, ker se dogaja, da pošta vrača pošiljke, ki uporabljajo stari naslov. Novi naslov je:

Časopis Informatica, Iskra Delta,
Stegne 15B, 61000 Ljubljana.

raziskovalni, razvojni, proizvodni, finančni in prodajni - ste nosilci in vlečni konji visokotehnološkega podjetja. Prav zato naj bi tudi vsi drugi - vsi delavci Iskre Delte in uporabniki Deltinih proizvodov - vlekli v vami in ne proti vam[6].

Današnja otvoritev računalniške tovarne in raziskovalno-razvojnega centra ob njej je praznik širšega okolja neglede na to, kako se to v tem trenutku manifestira[7]. To je zgodovinski korak v domači industrializaciji računalništva. Zato končujem s spoštovanjem, priznanjem in z zahvalo vsem, ki so na tej dolgi poti načrtovali, pripravljali in nosili kamne gradnike[8].

Hvala!

---

[7] Manifestacija otvoritve računalniške tovarne Iskre Delte je pokazala značilno nerazumevanje in s tem *socialno izoliranost* slovenskega univerzitetnega, znanstveno-institucionalnega, gospodarsko-zbornicnega in političnega estalishmenta do dejanskega tehnološkega razvoja in s tem do bistvenih preživetvenih možnosti slovenske in z njo vred jugoslovanske populacije. Kljub prisotnosti najvišjih predstavnikov slovenske znanosti in umetnosti, njenega predsednika akademika Janeza Milčinskega, visokih vojaških osebnosti in vrhunskih posameznikov strok, je lahko Mija Repovž naslednjega dne resignirano ugotovila:

"Če bi pomen neke tovarne presojali po gostih iz političnega vrha, ki naj bi s svojo navzočnostjo otvoritvenemu ceremonialu dali pečat družbene skrbi in pozornosti, je bila za ta podalpski kot jeseniška jeklarna dogodek številka 1 zadnjih desetletij, novi razvojno-proizvodni center Iskre Delte pa epizodica, ki naj ne moti velikega duha ..."

(Neelastična in soft pravila. Delo, stran 2, 5. decembra 1987).

---

[8] K načrtovanju, pripravljanju in nošenju kamnov gradnikov bi morali dodati še trpljenje, tragično dinamiko in dramatizacijo epopeje hitre rasti podjetja Iskre Delte in same graditve tovarne in njenih brezizhodnih posledic. Vrsta vodilnih delavcev je bila postavljenih v iskanje izhodov in je občutila nečloveške, brezobzirne in skrajno nehumane (totalitarne) pritiske, pod katerimi je doživljala tudi svojo lastno deformacijo in razcep. Doživljala je svojo lastno nehumanost in izgubljenost (suicidnost). *Mija Repovž*[9] priznava *Janezu Skrubeju*, generalnemu direktorju Iskre Delte vlogo jugoslovanskega Stevena Jobsa. Osebno sem prepričan, da bi vsak direktor, ki je s svojo osebno odgovornostjo zgradil tako veliko računalniško podjetje in s svojo širino omogočal razmah vrhunskih industrijskih raziskav in razvoja[10], moral imeti na voljo vsaj en izhod, ki bi bil priznanje (ne nagrada) njegovemu požrtvovalnemu, izčrpljajočemu delu, v katerem nikoli ni videl tiste lastne koristi, ki so jo lahko zaradi njegovega dela pobirali drugi. Skrajno neetično in za tehniško inteligenco nesprejemljivo je stališče, da lahko določena višja struktura, popolnoma birokratsko, post-totalitarno, po logiki svoje totalitarne samopoklicanosti stisne ustvarjalno osebnost v poligon svojih brezobzirnih in rušilnih manipulacij.

---

[9] Mija Repovž: Računalništvo na naš način. Delo, Sobotna priloga, stran 18 (5. decembra 1987).

---

[10] A. P. Zeleznikar: Parsys Expeditions to New Worlds. Informatica 11 (1987), No. 3, 76-80.

V tem svojem članku sem opozoril na posebno, seveda odločujočo, strateško smiselno vlogo generalnega direktorja Iskre Delte, Janeza Skrubeja: "... Quite at the beginning, Iskra Delta has introduced and incorporated strategic thinking and acting in its managerial decision making. As a fast growing enterprise in the field of computer industry, it has been confronted not only with the very basic organizational and technological problems of modern computer industry of the developed hemisphere, but also with specific problems of a technologically and even civilizationally (socially, ideologically) underdeveloped environment. In these times, up to this day, the general director of Iskra Delta

- Mr. Janez Skrubej -

was not only the real strategist of the company, but also the optimistic fighter, organizer, believer of the progress, and the carrier of several hard, arduous, and exhausting business situations and developmental processes of the company. And all of this in irregularly and unforeseeably changing circumstances of the underdeveloped hemisphere. Thus, it is to say, that *he* was the main initiator and supporter of the innovative- and independent Parsys project.

UDK 519.226.3

Matjaž Gams, Matija Drobnič
Jozef Stefan Institute

ABSTRACT. One of the recently developed systems for machine learning (GINESYS) significantly outperformed all compared systems including theoretically optimal Bayesian classifier, which was the second in both tests. We tested several options in Bayesian classifier to investigate the real cause for nonoptimal results and to estimate the upper limit in classification accuracy. The conclusion is that while it is possible to achieve even higher classification accuracy with suitable parameter adjustment in Bayesian classifier, it seems that GINESYS practically achieved the optimal classification accuracy.

Sistem za empirično učenje GINESYS je v praktičnih meritvah presegel primerjane sisteme z Bayesovim klasifikatorjem vred. Podrobna analiza kaže, da je dosežene rezultate mogoče preseči, vendar so že zelo blizu optimalne meje.

1. INTRODUCTION

Machine learning is a quickly developing area of Artificial Intelligence [Winston]. According to the major inference type used it can be divided into rote learning, learning from instruction, learning by deduction, by analogy, from examples and from observation and discovery [Carbonell et al; Michalski]. The scope of this article is learning from examples or Empirical Learning (EL). The aim of EL is to induce general descriptions of concepts from examples (instances) of these concepts. Examples are usually objects of a known class described in terms of attributes and values. The final product of learning are symbolic descriptions in human understandable forms. Induced descriptions of concepts, representing different classes of objects, can be used for classifying new objects. EL systems basically perform the same task (classification) as statistical methods and can be directly compared to them from the point of classification accuracy. On the other hand, EL systems offer further advantages, namely a) explanation during classification of new examples and b) the insight into the laws of the domain by observing classification rules. Explanation during classification (a) is important since it enables the user to check the line of reasoning and verify the system's decision. The knowledge base (b) can be viewed as a new representation of the domain knowledge, which can be of great value to domain experts, especially in domains that are not yet well formalized and understood.

2. A SIMPLE EXAMPLE

For a simple example let us consider a case where we have a device with 8 binary switches representing 256 legal combinations. Device reports errors in some combinations and we want to find out what subsequence causes them.

	SWITCHES								STATUS
	1	2	3	4	5	6	7	8	
1	0	1	1	0	1	0	0	1	ERROR
2	1	0	1	1	1	1	1	0	OK
3	0	0	1	0	1	1	0	1	ERROR
4	1	0	1	1	0	0	1	1	OK
5	1	0	1	1	1	0	1	1	ERROR

Table 1. Device reports error in some combinations of switches. Which subsequence of switches causes them?

Probably the most common answer in EL systems would be, that error is reported when switches 5 and 8 are on (=1).

In practical tasks EL systems deal with domains with 10 to 10.000 examples (typically some hundred) with 2 to 500 attributes (typically ten or some ten) [Breiman et al; Quinlan; Lavrac et al]. Attributes can be real, integer or categorical with many possible values.

3. EMPIRICAL LEARNING

The whole process of empirical learning consists of four steps:
- preprocessing of learning examples,
- construction of a classification rule,
- classification of new instances and
- analysing the laws of the domain.

Detailed description can be found elsewhere, e.g. [Kononenko] or [Gams, Lavrac] with detailed overview of some well known algorithms - C4 [Quinlan], CART [Breiman et al], ASSISTANT 86 [Cestnik et al], CN2 [Clark, Niblett], AQ15 [Carbonell et al]. We shall formally represent here only a domain area and a classification rule.

A set of learning examples L = {(x,c)} consists of pairs (x,c), where x is a vector (denoting properties of the object) in a

measurement space X and c represents the index of the class of example x.

Components of vectors x are called attributes or variables. The values of attributes can be numerical or categorical.

A classification or a decision rule d(x) is a mapping which maps every x from X into some c from C or into the probability distribution (p1,p2...pJ) where pi is a real number between 0 and 1.

A classification rule d(x) splits the whole space X into spaces X1,X2,...XJ, such that for every Xi only a certain subset of d(x) is relevant.

The syntax of a classification rule d(x) is:

```
<d>      ::= <Rule> | <Rule> and  <d>
classification rule

<Rule>   ::= <Class> if  <Cpx>
rule

<Cpx>    ::= <Sel> | <Sel> and  <Cpx>
complex

<Sel>    ::=   | Atr <op>  <Values>
selector

<Values> ::= Val | Val or  <Values>
values

<Class>  ::=  1|2|3...|J
class

<op>     ::= < | = |  >
operators
```

Atr corresponds to the name of the attribute and Val is a categorical or numerical value.

This syntax is transformable into DNF and is similar to the syntax of most rule-based systems or expert systems [Waterman, Hayes-Roth]. Note that the actual syntax is slightly more complicated [Gams].

4. DOMAIN DESCRIPTION

We performed practical measurements on two real-world domains. Data were obtained by I. Kononenko and represent descriptions and diagnoses of patients from the Oncological Institute Ljubljana. The only correction was replacement of missing values by the most probable ones for a given class. More detailed description is in [Gams], here we present only cumulative data about these domains:

Domain 1

```
number of attributes                        18
no. of possible values per attribute     2 - 8
                                 ( average 3.3 )
number of classes                           9
total number of examples                  150

distribution of examples amongst classes
number of examples in C1 to C9
1  2   3  4   5  6  7  8  9
2  1  12  8  69 53  1  4  0

importance of attributes - A1 to A18 :  none
of them is redundant
```

Domain 2

```
number of attributes                        17
no. of possible values per attribute     2 - 3
                                 ( average 2.2 )
number of classes                          22
total number of examples                  339

distribution of examples amongst classes
number of examples in C1 to C22
 1  2  3  4  5  6  7  8  9 10 11
84 20  9 14 39  1 14  6  0  2 28

12 13 14 15 16 17 18 19 20 21 22
16  7 24  2  1 10 29  6  2  1 24

importance of attributes - A1 to A17
(counting how many examples overlap when
omitting the i-th attribute)
 1  2  3  4  5  6  7  8  9
80 80 58 85 60 53 65 55 68

10 11 12 13 14 15 16 17
63 55 60 53 54 57 65 65
```

5. GINESYS

5.1. ALGORITHM DESCRIPTION

The top level description of GINESYS (Generic INductive Expert SYstem Shell) is as follows:

```
repeat
  initialize Rule;
  generate Rule;
  add Rule to d(x);
  L = L - {examples covered by Rule}
until satisfiable(d(x))
```

In this general view GINESYS represents a prototype of a unifying algorithm for empirical learning covering many other systems. In a slightly more specified description we obtain the following algorithm:

```
repeat
  generalize Rule;
  repeat
    specialize Rule
  until stop(Rule);
  postprocess(Rule);
  add Rule to d(x);
  L = L - {examples covered by Rule}
until satisfiable(d(x))
```

The main difference between other EL systems and GINESYS is in "confirmation rules". Basic idea of confirmation rules is using several sources of information for classification. That seems to be common practice in every day life. For example when we try to predict the weather, we look at the official weather report, but also look at the sky and ask our neighbour. The implementation of this idea in GINESYS is that instead of using only one rule for classification several rules confirm or confute the first one. In case of a confrontation between these rules the Bayesian classifier is consulted as a method of a conflict resolution [Waterman, Hayes-Roth]. One confirmation rule in our simple example in Table 1 could be : Error is reported, when switches 3, 5 and 8 are on. This rule could be redundant or even wrong, but on the other hand it could be the only correct one! From examples in Table 1 it is not clear which of these possibilities is the right one, so both (and other) rules are

stored and consulted. In more detailed tests [Gams] it was shown that this method of consulting several rules (= using different kinds of information) significantly improved classification accuracy.

5.2. COMPARATIVE RESULTS

A detailed comparison was made with other well known EL systems in two noisy medical domains (oncology). Table 2 shows results in classification accuracy.

	domain 1	domain 2
GINESYS	69.9	51.9
BAYES	68.4	50.1
other systems	67.3	48.7

Table 2. Classification accuracy measured as the percentage of correct guesses.

While GINESYS achieved best results and Bayesian classifier the second ones, none of the compared systems [Gams] outperformed results in the last row in Table 2. These results are actually an average over ten runs on randomly chosen 70% of data for learning and remaining 30% of data for testing. In further tests (t-tests, [Gams]) it was shown that the number of tests, distribution and the difference between classification accuracies was sufficient to ensure that differences are a result of some deeper cause (e.g. better algorithm) and not a chance choice.

Other measurements proved superiority not only from the point of classification accuracy, but also in generality, complexity of classification rule and explanation [Gams]. GINESYS and other algorithms discussed in this paper were implemented in Pascal on VAX 11/750.

5.3. IRREPROACHABILITY OF MEASUREMENTS

We argue that our measurements are irreproachable (unbiased) since:

- all systems were measured on exactly the same data
- no "cleaning" of data was performed
- no special form of data was allowed
- no unusual method of measuring classification accuracy was used
- no domain dependent parameters were allowed
- the number of data and tests was sufficient (t-tests) to avoid chance choice
- results were strictly checked and verified by many supervisors from the program source level to the level of classification trace.

5.4. DISCUSSION ABOUT RESULTS

Some of the systems for empirical learning achieved good results in practical testing in several real life domains, practically approaching or even outperforming domain experts and statistical methods [Kononenko; Michalski, Chilausky; Breiman et al; Gams]). More acceptable is the opinion [Breiman et al], that although all methods are more or less domain dependent, EL systems in general achieve about the same classification accuracy as other statistical methods. In our measurements some of the EL systems, especially those without special mechanisms for noisy domains, gave unexpectedly poor performance compared to the results published by the originators of algorithms. Since our implementations of those systems were the same as published, several possible explanations remain. It might be that actual implementations use some unpublished extra features, maybe the domains used for testing were especially suitable for specific algorithms etc.

The authors of this article also find questionable comparing between the system and the expert, since we regard EL systems mainly as a helping tool and not as a stand-alone program. The other reason is that fair comparison between machine and human is extremely difficult. The correct comparison should be (system + user) : user.

In most complex realistic domains mechanisms for dealing with noise are of greatest importance as independently discovered in [Breiman et al; Kononenko] and it is not realistic to achieve even tolerable results without them [Kononenko; Gams].

6. BAYESIAN CLASSIFIER

6.1. THEORETICAL FOUNDATIONS

The concept of the Bayes rule is one of the most important concepts in the field of classification and also learning. For the data drawn from a probability distribution P(A,j), the most accurate rule can be given in the terms of P(A,J) and this rule is called the Bayes rule. It is normally denoted by dB(x).

Precisely, suppose that (x,y), $x \in X$, $y \in Y$ is a random sample from the probability distribution P(A,j) on X x C, i. e., $P(x \in A, y=j)=P(A,j)$. Then we define dB(x) as the Bayes rule if for any other classifier d(x),

$$P(d_B(x) \neq c(x)) < P(d(x) \neq c(x))$$

Let us assume that X is N-dimensional euclidean space and for every j, j=1, ... ,J, P(A/j) has the probability density fj(x) and for sets $A \subset X$

$$P(A/j) = \int_A f_j(x)dx$$

Then we can prove the following theorem:

$$d_B(x) = (j\,;\, f_j(x)P_e(j) = \max_i f_i(x)P\,(i))$$

where J is the number of classes and P(j) is the prior probability of the class j. The proof can be found in [Breiman et al].

In practice, neither the P(j) nor the fj(x) are known. The three most common classification procedures, used to approximate the Bayes rule by using the learning sample data, are discriminant analysis, kernel density estimation and k-th nearest neighbour. Accuracy of two of them have been compared with the results of Ginesys on both domains.

6.2. PRACTICAL IMPLEMENTATIONS

The k-th nearest neighbour method [Fix, Hodges] was implemented as simple as possible. The algorithm searches through the set of learning examples and determinates distance between learning and test example as the number of mismatches in their attribute values. Test example is then classified by its first nearest neighbour, and if there are more equally distant neighbour, the last one found is picked for classification. It is so called Nearest-neighbour classifier [Batcheles 1974]. Although it is so primitive, this method classifies test examples with 72.9% average accuracy in the domain 1, what is even better than GINESYS. However, in the domain 2, which is far more complex, the classification accuracy is only 40.4% what is considerably lower than that of the other methods.

The following approximation of the Bayes rule [Clark, Niblett; Kononenko] is one of the most commonly used. In general, the rule is formed as

$$P(c/A) = P_a(c)\frac{P(A/c)}{P(A)} = P_a(c)\frac{P(\bigwedge A_i/c)}{P(\bigwedge A_i)}$$

At this point the assumption is made, that all attributes are independent :

$$P(c/A) = P_a(c)\frac{\prod_i P(A_i/c)}{\prod_i P(A_i)} = P_a(c)\frac{\prod_i P(A_i/c)}{\prod_i P_a(A_i)} \qquad (1)$$

When classifying a new example we need to evaluate formula (1). One practical solution when dealing with categorical values is to store all factors into a 3-dimensional table TB[i,j,k] with the following indexes :

i - attribute index
j - attribute's value's index
k - class index

TB[i,j,k] is the number of examples in the learning set with the properties, denoted by index values.

When evaluating formula (1) during classification of a new example, one of the factors can be 0. The result can be either 0 or undefined. The solution in the second case is obvious - delete this attribute from the formula. The same solution is sometimes used when the result is 0.

In Table 2 GINESYS (without domain dependent parameters or other adjustments [Gams]) achieved higher classification accuracy than the practical implementation of theoretically optimal Bayesian classifier. The reason for this must be in practical implementation, especially in

a) approximation of probabilities from the learning set,
b) assumption, that attributes are independent,
c) practical solutions to numerical problems.

Problem (a) can be discarded, since all systems [Gams] processed exactly the same data. But it could be the case, that different classifiers (also different implementations of Bayesian classifier) are more and other less sensible to the number and distribution of input data.

6.3. A PRACTICAL EXAMPLE

Problems appearing during the evaluation of formula (1) can be shown in a simple example. Let us try to classify examples e1 and e2 from data obtained from Table 1.

e1 = 0 0 0 0 0 0 0 0
e2 = 1 0 1 1 1 0 1 1

$$P(OK/e1) = \frac{2}{5}\cdot\frac{0}{2}\cdot\frac{2}{3}\cdot\frac{0}{2}\cdot\frac{0}{2}\cdot\frac{1}{3}\cdot\frac{1}{2}\cdot\frac{0}{2} = ?(2.6)$$

$$P(ERROR/e1) = \frac{3}{5}\cdot\frac{2}{3}\cdot\frac{2}{3}\cdot\frac{2}{3}\cdot\frac{2}{3}\cdot\frac{2}{3}\cdot\frac{2}{3}\cdot\frac{0}{3} = ?$$

$$P(ERROR/e2) = \frac{3}{5}\cdot\frac{1}{3}\cdot\frac{2}{3}\cdot\frac{1}{3}\cdot\frac{2}{3}\cdot\frac{2}{3}\cdot\frac{1}{3}\cdot\frac{2}{3} = 0.149$$

$$P(OK/e2) = \frac{2}{5}\cdot\frac{2}{2}\cdot\frac{2}{3}\cdot\frac{2}{2}\cdot\frac{1}{3}\cdot\frac{1}{2}\cdot\frac{2}{3}\cdot\frac{1}{2} = 0.754$$

None of the two examples gives the sum of all classes equal to 1. Even if we delete all columns having 0 we obtain results like P(OK/e1) = 2.6. Furthermore, in case of example e2 the probability for class OK is greater than for the class ERROR, although example e2 is the same as example e5 from Table 1, belonging to class ERROR. However note that the nearest neighbour method would classify correctly in this case.

A small number of examples is insufficient for most statistical methods and also for Bayesian classifier. In real measurements in domain 1 and 2 the number of examples was always greater than one hundred and was considered sufficient. Nevertheless these counterexamples show that better classification accuracy is possible.

7. ADJUSTING PARAMETERS IN BAYESIAN CLASSIFIER

Probabilities used in evaluating formula (1) are approximated by prior probabilities in the learning set, what yields some error in classification. The formula is then evaluated as follows

$$P(c/A) = P_a(c) * \frac{\prod_i \frac{a_i}{b}}{\prod_i \frac{c_i}{d}} \qquad (2)$$

where

- ai is the number of examples of the class c with the same value of the i-th attribute as the test example,
- b is the number of examples of the class c,
- ci is the number of all examples with the same value of the i-th attribute as the test example, and
- d is the number of all learning examples.

When dealing with noisy data, errors may occur during evaluation of formula (2). Two methods have been used to avoid this errors.

7.1. OMITTING THE UNRELIABLE FACTORS

The main idea of this method is that the accuracy of estimations in formula (2) grows with the number of examples. Therefore, if b

or d (only b in practice) is smaller than parameter MINN, which is set before evaluation, then the factor with this b is omitted during evaluation of formula (2) and the class probability is estimated by its prior probability. Table 3 shows the results in classification accuracy with different values of MINN.

MINN :	domain 1	domain 2
0	68.4 (0.0)	50.1 (0.0)
1	68.4 (11.1)	50.1 (4.5)
2	68.4 (47.8)	49.7 (31.8)
3	69.3 (52.2)	49.7 (32.3)
4	68.7 (56.7)	49.8 (33.6)
6	68.0 (60.0)	50.9 (47.7)
10	66.7 (75.6)	48.7 (60.5)
15	67.1 (77.8)	46.2 (72.7)

Table 3. Classification accuracy measured as the percentage of correct guesses. Values in brackets are percentages of classifications with prior probability.

It is interesting that accuracy is almost independent of the number of classifications with prior probabilities and it decreases only if we classify approximately 75% of examples this way. Yet we can see that accuracy on both domains reaches its maximum when approximately 50% of classifications are done by the prior probability of classes and this maximal accuracy is near the accuracy of GINESYS.

7.2. ADJUSTING ZERO FACTORS

A problem occurs, what to do when ai in the formula (2) is 0. One solution is to omit this factor from the formula. Another idea could be to set ai to a very small number EPS in such case. The idea is that this zero is the result of domain noise and, with more learning examples, we would sooner or later find such example and therefore we made almost no mistake and we also don't lose the information contained in the distribution of other attributes. The results of this method are shown in Table 4.

EPS :	domain 1	domain 2
0.00	68.4	50.1
0.01	68.7	52.8
0.05	71.1	52.9
0.10	72.2	51.9
0.50	24.2	27.7
1.00	2.2	7.5

Table 4. Classification accuracy achieved by adjusting zero factors in formula (2) by some small value EPS.

In both cases this method achieves accuracy higher than GINESYS. But, rapid drop of accuracy also shows that this method is very sensible to the value of EPS. Whenever we set zero factor to some value different from 0 we introduce an error into the evaluation of formula (2) and if the value of EPS is too big the results are no more reliable at all.

7.3. COMBINATION OF BOTH METHODS

In this case, both methods described in 7.1. and 7.2. are used together. First we look whether b and d are bigger than MINN and than we set zero factors in formula (2) to EPS where needed. The results of measurements on both domains are shown in Table 5 and Table 6.

\ EPS : / MINN :	0.00	0.01	0.05	0.10
0	0.0	0.3	2.7	3.8
2	0.0	1.4	1.1	1.1
4	0.3	1.2	0.9	0.3
6	-0.4	0.0	-0.4	-0.4

Table 5. Increase of classification accuracy by combination of both methods in domain 1 (basic accuracy 68.4%).

\ EPS : / MINN :	0.00	0.01	0.05	0.10
0	0.0	2.7	2.8	1.8
2	-0.4	1.9	2.2	2.0
4	-0.3	1.8	2.1	2.0
6	0.8	2.7	3.2	3.2

Table 6. Increase of classification accuracy by combination of both methods in domain 2 (basic accuracy 50.1%).

The accuracy of GINESYS is in both cases exceeded by more than 1% what is also the difference between basic Bayesian classifier and GINESYS. Yet there is a problem which combination of EPS and MINN values to choose and how far this decision is domain independent. Therefore, GINESYS can still be considered to reach the practical upper bound of classification accuracy.

8. EXTERNAL RULE DIRECTED BAYESIAN CLASSIFIER

Bayesian classifier itself does not derive any explicit rules and therefore rules generated by some other system (in our case GINESYS) can be used to control the evaluation of formula (1). Two such methods have been tested. The idea of the first one is that any (more or less successful) rule denotes a complex of attributes which are logically connected and therefore a deviation from the optimal Bayesian classifier is somewhat corrected. The second one is an attempt to cross GINESYS and Bayes together to yield better results.

8.1. CLASSIFICATION WITH IMPORTANT ATTRIBUTES

This method uses rules, generated by GINESYS. During evaluation of formula (1) the rule which matches the current example is searched for. If it is found, we calculate adequate probabilities by searching through the table for entire complex and not by decomposing the attribute complex to basic attributes. On the other hand, if the matching rule is not found, undisturbed evaluation of formula (1) follows. The results are shown in Table 7.

Measurements show that introducing of externally generated rules into Bayesian classifier only slightly disturbs its classification accuracy.

8.2. CLASSIFICATION WITH IMPORTANT ATTRIBUTES ONLY

The main idea of this method is to use important attribute complexes in classification if possible. For each example classified we first search for the matching GINESYS rule. If such one is found, it is used for classification. If not (when only Null rule of GINESYS is found), the

classification is carried out by formula (1). The results are shown in Table 7.

	domain 1	domain 2
Bayes	68.4	50.1
Rule Directed Bayes	68.4	49.2
Important Attr. & Bayes	69.3	47.9

Table 7. Classification accuracy measured as the percentage of correct guesses.

9. COMPARISON BETWEEN EL SYSTEMS AND STATISTICAL CLASSIFIERS

Let us summarize conclusions from previous paragraphs:
- It is possible to further improve classification accuracy of implementations of Bayesian classifier, even to overpass best results of GINESYS.
- Among compared methods without domain-independent parameters GINESYS performs best and is very close to the practical limit in classification accuracies in measured domains.

Statistical classifiers are basically unable to perform explanation during classification and to build a human understandable knowledge base. Besides these, other disadvantages can be pointed out [Breiman et al; Gams]:
- they can not deal with domains with small number of learning examples;
- it is difficult to deal with unusual situations (deleting by 0, unknown values, ...);
- their results vary according to the suitability of problem domain.

It is only fair to notice that more advanced statistical methods eliminate some of these disadvantages. However these properties remain basically unchanged.

Another reason for so good results of EL systems like GINESYS compared to statistical methods is shown in Figure 1. Real-life complex domains probably contain logical laws which cover greater areas regardless of noise in given examples. On the contrary statistical methods depend on variations of probability distribution. In Figure 1 the correct probability distribution for classes 1 and 2 is presented by bold lines. Dotted lines represent probability distribution, obtained from given examples. Because of the fact that probability distribution is more sensible to chance choice it is possible that dotted lines 1 and 2 overlap causing incorrect classification.

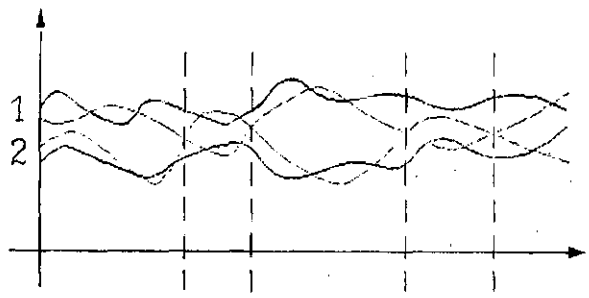


Figure 1: A graphical representation of one of the possible reasons why GINESYS performs so well compared to statistical methods.

10. CONCLUSION AND DISCUSSION

Older systems for empirical learning (EL) outperformed the statistical methods from the point of explanation during classification and possibility of building a human understandable knowledge base. While it was in some cases reported that older EL systems outperformed statistical methods as well as domain experts this opinion is not undoubtedly shared with the authors of this article. More acceptable is the conclusion [Breiman et al], that the best EL systems achieve about the same classification accuracy as statistical methods. Nevertheless it seems that the new breed of EL systems with GINESYS as one of the most promising representatives outperforms statistical methods even in classification accuracy (at least in so far measured domains).

ACKNOWLEDGES

We are grateful for suggestions to prof. Ivan Bratko. Marjan Petkovsek provided mathematical background for our analysis. Students Izidor Jerebic, Borut Znidar, Aram Karalic and Darko Zupanic were of great help in programming tasks. Igor Kononenko and the Oncological Institute Ljubljana enabled us to do the measurements on real-world domains. This work was partly supported by the Slovene Research Council and COST-13. Research facilities were provided by the "Jozef Stefan" Institute.

REFERENCES

Breiman L., Friedman J.H., Olshen R.A., Stone C.J. (1984): "Classification and Regression Trees", Wardsworth Int. Group.

Carbonell J.G., Michalski R.S., Mitchell T.M. (1983): "An Overview of Machine Learning", in Michalski R.S., Carbonell J.G., Mitchell T.M. (ed.), Machine Learning: an Artificial Intelligence Approach, Tioga Publishing.

Cestnik B., Kononenko I., Bratko I. (1987): "ASSISTANT 86: A Knowledge Elecitation Tool For Sophisticated Users", Progress in Machine Learning, Sigma Press.

Clark P., Niblett P. (1987): "Induction in Noisy Domains", Progress in Machine Learning, SIGMA Press.

Kononenko I. (1985): "Razvoj sistema za induktivno učenje ASISTENT", magistrsko delo, Fakulteta za eletrotehniko, Ljubljana.

Kononenko I. (1985): "Strukturno avtomatsko ucenje", Informatica 3, str. 44 - 56.

Gams M. (1987): "Principi poenostavljanja v sistemih za avtomatsko ucenje", doktorska disertacija, Ljubljana.

Gams M., Lavrač N. (1987): "Review of Five Empirical Leaning Systems Within a Proposed Schemata", Progress in Machine Learning, (ed. Bratko I., Lavrac N.), Sigma Press.

Lavrač N., Varšek A., Gams M., Kononenko I., Bratko I. (1986): "Automatic construction of the knowledge base for a steel classification expert system", The 6th International Workshop on Expert Systems, Avignon.

Michalski R.S. (1987): "Machine Learning", Tutorial 6, IJCAI 1987, Milano.

Michalski R.S., Chilausky L.R. (1980): "Learning by Being Told and Learning from Examples: an Experimental Comparison for Soybean Disease Diagnosis", Policy Analysis and Information Systems, Vol. 4, No 2.

Quinlan J.R. (1986): "Induction of Decision Trees", AI Summer Seminar, Dubrovnik.

Waterman D.A., Hayes-Roth F. (ed.) (1977): "Pattern-Directed Inference Systems", Academic Press.

Winston H.P. (1984): "Artificial Intelligence", Addison-Wesley.

UDK 681.3.083.7

Niko Guid, Borut Žalik
Tehnical Faculty Maribor

Abstract: In this paper standards in computer graphics are described. At first the reasons for evolution these standards are given and then the ways of accepting the international standards are presented. Afterwards the evolution phases of the graphical standards under ISO and ANSI are interpreted and current stage of particular standards are given. In proceeding the place of graphical standards and standard proposals in a graphical system are shown. Finally, the position and role of the graphical standards in a modern CAD system is presented.

Povzetek: V članku podamo pregled standardov v računalniški grafiki. Najprej opišemo vzroke za razvoj teh standardov, nato pa prikaženo poti, preko katerih nek predlog lahko postane mednarodni standard. Zatem predstavimo razvojne faze ISO in ANSI standardov ter podamo trenutne razvojne stopnje posameznih standardov za računalniško grafiko. V nadaljevanju opišemo mesto grafičnih standardov oziroma predlogov standardov v grafičnem sistemu. Nazadnje podamo mesto in vlogo grafičnih standardov v modernem CAD sistemu.

1. THE BEGINNINGS OF STANDARDS DEVELOPMENT

Today, a large number of different graphical hardware and even more different graphical software exist. A big part of this graphical software is **device-dependent**. The consequences are:

1. It is impossible to exchange graphical software between different graphical systems
2. There are problems by installation of old programs on new graphical equipment, although it has been suplied by the same producer, etc.

Because of these problems an idea has been appeared to make a device-independent graphical packet. Advantages of this **device-independent graphical packet** are:

1. It could serve different device generations.
2. Programs could work on different graphical systems.
3. Programmers could immediately work on different graphical systems.
4. Graphical systems are distinguished only by quality, price, and efficiency.

Of course, this device-independent graphical packets have also some weaknesses. **They are slower** than device-dependent and **more memory space is needed.** Because the power of computers is rapidly increasing and their prices are decreasing, advantages of standardization are going over its weaknesses. People, who are opposite to standards in computer graphics, affirm **that standards are against inovations.** It is clear, when a standard is accepted, it could not be changed immediately.

Portability of aplication programs could be achieved in some different ways /ENDE84/:

- with development of computer languages,
- with extension of existing program languages with graphical features or
- with libraries of graphical subroutines which could be linked into application program.

Experts from the field of computer graphics have chosen **the last possibility** by the construction of international device-independent graphical standard. However, it is least elegant of all but it is the best way to awoid confusing in structures of program languages. The place of the device-independent graphical standard in a graphic system is shown by figure 1.

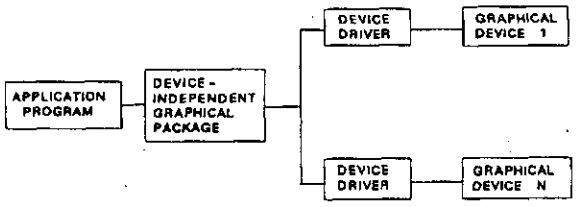


Figure 1. The place of graphical standard in graphical system

The development of graphical standards began in the year 1974, when the **Graphics Standard Planing Committee (GSPC)** was found by **ACM SIGGRAPH** ("Association for Computing Mashinery Special Interest Group on Graphics"). This committee met with other international members, involved in computer graphics in Seillac (France) in 1976. This meeting had a great influence on first draft standard called **Core System**. It was introduced on SIGGRAPH 1977. Two years later, on SIGGRAPH in 1979, an improved version of Core was appeared.

Soon after that a new group has been found by **German Institute for Standards DIN** which has been worked on a new graphical standard basis. The group has been directed by Jose Encarnacao and **it prepared in 1977 a draft standard called GKS** (Graphical Kernel System).

Two propositions were apeared by ISO in 1979: Core and GKS. Working Group WG2 by ISO decided that only efforts on GKS continued. GKS was much more simple, it was 2D, and it was intended for raster devices. On other side Core was 3D and destined for vector devices. The first draft proposal of GKS was made by ISO in 1982. **GKS was accepted as an ISO standard in 1985.**

In 1981 SIGGRAPH GSPC committee was disbanded and passed over to the ANSI X3H3 committee, which was founded in 1979.

GKS has become a basis for many other proposals of standards including **PHIGS, CGM, and CGI.**

IGES, as a standard for transferring CAD/CAM data bases, has been developed in completely another way than Core and GKS (through others ANSI committees). **IGES was accepted as an ANSI standard in 1981.**

2. WHO SETS UP THE STANDARDS?

American National Standard Institute (ANSI) does not set up the standards, but it only whaches over the process, through which the standards are accepted. ANSI has to notice if a standard draft is acceptable by most wide part of industry. Only such standard could be adopted and used in industry and other institutions. ANSI adopts a standard as a national standard when it is acceptable by most companies and organizations.

ANSI consists of by several committees. So the **ANSI X3 is the standards development committee for information processing** and has about 30 committees, each with about 15 to 80 members /BONO86/. One of them is **X3H3 tehnical committee, which is responsible for computer graphics standards.** X3H3 committee consists of 6 subcommittees, which is showed by figure 2 /STRA86/.

More than 100 participants, representing about 80 companies (CalComp, Control Data, DEC, HP, Honeywell, IBM, Intel, Tektronix, TI, etc.), attend X3H3 meetings.

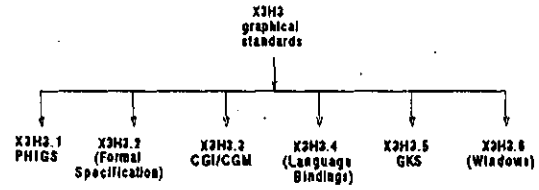


Figure 2. X3H3 Tehnical Committee and its subcommittees

It is similar for **International Organization for Standardization** (ISO). ANSI is only a secretariate in ISO's Technical Committee TC97. Working Group WG2 in subcommittee SC21 is responsible for graphical standards. Its sign is ISO/TC97/SC21/WG2.

Some standards which are set up by ISO or ANSI, are effective, but others are even ignored. From this point of view standards could be considered as **de facto** and **formal standards** /STRA86/.

For example, IBM's Color Graphics Adapter (CGA) is a de facto display standard for PCs, just as the IBM PC is a de facto standard for personal computers. Neither CGA neither IBM PC was formal standards, but market factors has adopted them as standards.

Among formal standards we distinguish successful and unsuccessful ones. A case of formal standard, which has been widely used, is RS-232-C. On the other hand, who has heard about ANS X3.23 standard for keyboard layout. This standard has been totally eclipsed by de facto standards, first by the IBM Selectric layout and later by PC and AT layouts.

3. DEVELOPMENT PROCESS OF STANDARDS

That a proposal becomes a standard, there are several phases it must go through. It takes much more time for standardization process in computer graphics than for standards from other areas, because the projects are very large and completly new.

Evolution process of an ISO standard

Evolution phases of an ISO standards are /BONO85, BONO86/:

- the first: **New Work Item proposal** (NWI); discussion about new project is started, when subcommittee (like SC21) or a member body (like ANSI) makes a proposal. Representatives of different countries decide if they accept the definition of the work item and if the work is continuing on this proposal. This stage can take 5 to 8 months.
- the second: **Working Draft** (WD); document could be in this stage 6 to 18 months;
- the third: **Draft Proposed** (DP); this stage can take 12 to 14 months;
- the fourth: **Draft International Standard** (DIS); document could take place in this stage for 9 to 12 months;
- the final: **International Standard** (IS).

Evolution process of an ANSI standard

Evolution stages of an ANSI standard differ from stages of an ISO standard and are following:

- the first: **Standing Document 3** (SD-3) is an initial proposal which can take no less than 6 months;
- the second: **Working Drafts**; X3H3 prepares a series of working drafts that are circulated among X3H3 members. This stage typically takes several years.
- the third: **Draft Proposed American National Standard** (dp ANS); this stage takes 6 to 10 months;
- the fourth: **Public Review**; document could be in this stage 8 months or more, which depends on the number of public reviews. At least two public reviews are required by X3H3.
- the final: **Final Approval** takes 6 to 9 months.

The current stage of graphical standards under ISO and ANSI is shown by table 1 /BONO86, SELE87/.

Table 1. The stage of graphical standards project

Project	ISO status	ANSI status
GKS	IS 7942 published in Avgust, 1985.	ANS X3.123-1985. Published in October, 1985.
GKS Fortran	Known as ISO DIS 8651/1. DIS ballot closed in Avgust, 1986.	ANS X3.124.1-1985. Published in October 1985.
GKS Pascal	Known as ISO DIS 8651/2. DIS Ballot closed in August, 1986.	ANS X3.124.2-1987. Public review closed in May, 1987.
GKS Ada	Known as ISO DP 8651/3. Second DP ballot closed in April, 1986.	ANS x3.124.3-198x. Public review closed in 1986.
GKS C	Not yet an ISO standard language. WD available now (SC21/N669).	ANS X3.124.2-198x. Public review will be finished by October,1987.
GKS-3D	Known as ISO DP 8805. Second DP ballot closed in March 1986.	Public review finished in 1986.
GKS-3D Fortran	Known as ISO DP 8806.	Public rewiew finished 1986.
GKS-3D Pascal	Not yet available.	
PHIGS	WD finished in 1986.	ANS X3.144-198x. Second public review finished in 1987.
PHIGS Fortran	WD available. (SC21/N667)	Second public review finished in 1987.
PHIGS Ada	WD available. (SC21/N819)	Public review finished in 1986.
CGM (former VDM)	IS 8632 published in 1987.	ANS X3.122-1986 published in 1986.
CGI (former VDI)	DP began in 1986.	ANS X3.161-198x. Public review finished in Juny, 1987.

4. THE PLACE OF THE GRAPHICAL STANDARDS IN THE GRAPHICAL SYSTEM

Six known standards (suggested or accepted) could be devided into **three chategories** /DEUS84/:

1. **Core, GKS, and PHIGS** represent an **aplication programming interface** (API). This API standards are usually implemented as a set of the external procedures and an application programmer could link them into his application code.
2. **IGES and CGM** are used by **transfering and storing** the graphical information.
3. **CGI** represent an **graphical device interface.**

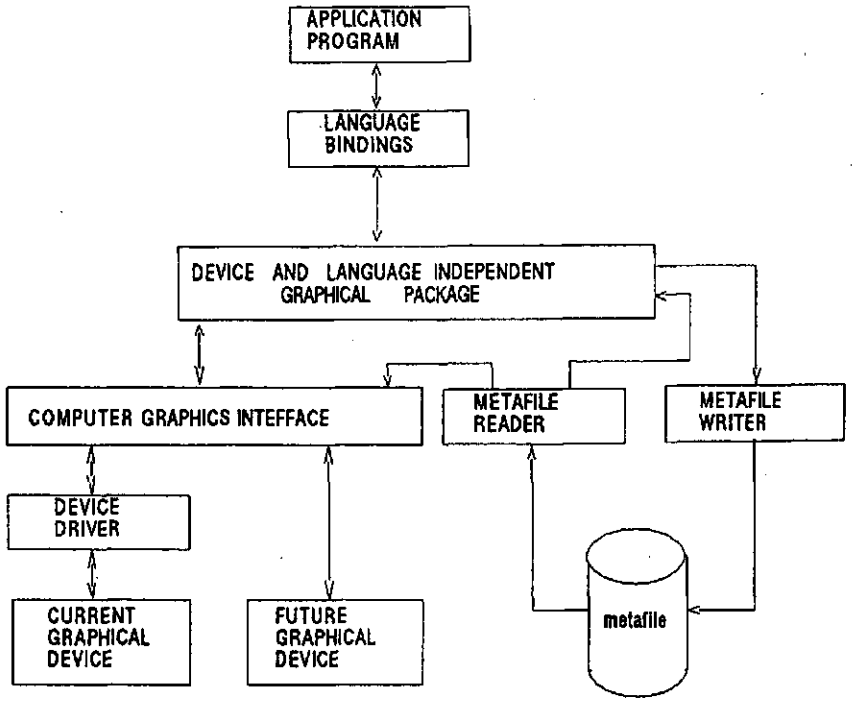


Figure 3. Interfaces of the graphical system

These tree classes could help us by defining common features of current and future standards with regard on performance, price, and usefulness of graphical software and hardware. A comparison and valuation of the graphical standards is not easy, because the majority of them are very complex and because they are comming from different areas. Figure 3 gives a review of the standard graphical interfaces.

The most important part of each language and device independent graphical system is a member of the GKS standards family. These are GKS (Graphical Kernel System) or its 3D extension or a standard for dynamical manipulation with graphical data structures PHIGS (Programmer's Hierarchical Interactive Graphics System). Functions of graphical system are exactly defined by these graphical standards and because of this reason they are also called **functional graphical packets.** They are completly language and device independent.

An aplication programmer is able to access functions of these packets through a **language binding.** It has to adopt language independent functions of graphical packet to design and particularities of each high level program language (ada, C, pascal, fortran, basic).

The communication between a language and device independent graphical packet and graphical workstations is controled by CGI (Computer Graphical Interface). This standard defines functions and format for this communication.

Current graphical devices are not able to receive the CGI format and interpret its functions directly yet, so drivers are needed. But **next-coming graphical devices** are going to be driven by CGI format directly.

In capturing, storing, and transfering of graphical information standard CGM is involved (Computer Graphical Metafile). Pictures are saved into **metafiles,** and are captured from functional graphical packet by **metafile generator.** The metafile contents is interpreted by **metafile interpreter.** Metafile could be interpreted directly by CGI or by functional graphical packet (GKS has particular types of workstations intend for manipulation with metafiles).

In presentation of interfaces of graphical system many authors mark also **connective links** between individual interfaces. They call them interfaces, too. Connective link presents a set of all functions, which the interface on the higher level of hierarhy of graphical system can access from the interface, which is lower.

For example: A connective link between a language binding and a aplication program is an exact declaration of all functions, which aplication programmer can include in his programs. There are declarations of all parameters and their types, which are used in the functions.

5. GRAPHICAL STANDARDS AND THEIR POSITION IN CAD SYSTEM

The most important aplication area of computer graphics is certainly CAD. Graphical system in the CAD system has to care about:

- the graphical presentation of constructed objects and
- about the graphical interaction with an user.

If figure 3 is extended for a CAD aplication, a structure of CAD system is reached and it is shown on figure 4 /ENDEB6/. Older CAD systems have been usually put into only one product. Individual pieces of them have been neither

evident neither accessible by the user. Trends of next-coming CAD systems are to make the graphical system visible also to the user, so that he could add different graphical devices and transfer graphical data between different graphical systems. But this is possible only, if a graphical system consists of the standard interfaces.

A core of a CAD system includes functions for modeling, presenting, calculating of constructed models, and a modul for interactive dialog with an user. The most important part of a CAD system is a CAD data base, which saves all information about created objects. A core of a CAD system **is only a superstructure** of the graphical system, which takes care about objects presentation on graphical devices and for graphical interaction with its standard interfaces. A modern CAD system has his own program's interface. So an user can reach the functions of the CAD system from high level program language and has an opportunity to solve and to present his own specific demands.

An exchange of data constructed by core of the CAD system (data are not only graphical) between systems of different supliers could be done by standard CAD data interface. The first such standard is **IGES** (Initial Graphics Exchange Specification), but just now more new exchange data formats is being developed (PDES, PDDI, SET, STEP) /CAD85, ENDE86, WILS87/.

6. CONCLUSION

All problems of device-dependent graphical software have been solved with introducing the graphical standards. The suppliers of graphical hardware and software are aware of this and today some of these standards are accessible even on PC computers (for example GKS level 2b).

In regard of development phases of graphical standards and their language bindings we can expect, that all GKS language bindings (with exception the language bindings for C, because it is not a standard language yet) will become the international standards in a short time.

The basical request for GKS-3D is fully compatibility with GKS. Currently, there are discusions about compatibility among PHIGS and GKS. It seems, that there will not be a compatibility at all, because GKS uses only *one-leveled graphical data structure* (segments), while on other hand PHIGS manages with hierarhical data structures, which are suitable for presenting the graphical models. PHIGS is intended for time-demand applications and so it more than likely will not be available on PC computers.

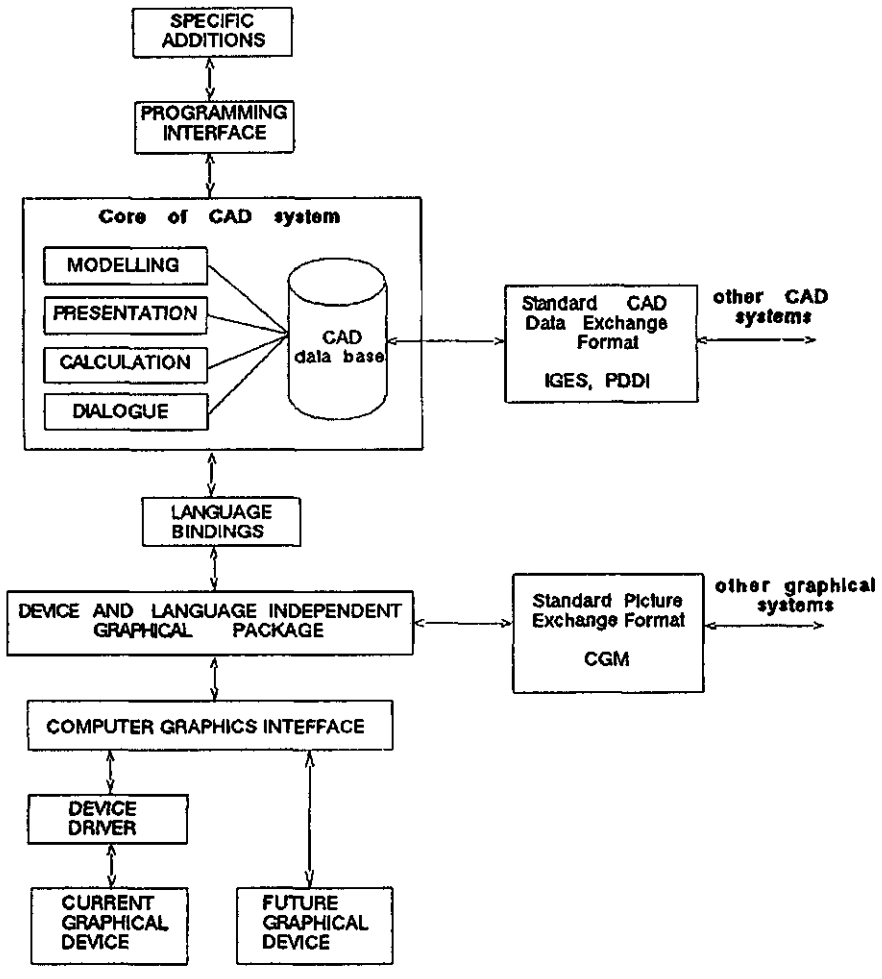


Figure 4. A place of the standards in CAD system

CGM has already become an international standard in its elementary version. The work is proceeding now on an expansion of CGM, that it could support also GKSM (GKS metafile) format (this is format in which GKS saves graphical information through logical workstation MO and MI).

The evolution of CGI has been already started. There is a long way until CGI as an international standard will be accepted, because this standard should not limiting the development of the graphical hardware. So we can expect a great interest and influence of manufacturers of the graphical hardware.

References

BONO85 Bono, P.R.,
"A Survey of Graphics Standards and Their Role in Information Interchange",
IEEE Computer, Vol. 18, No. 10,
October 1985

BONO86 Bono, P.R.,
"Guest Editor's Introduction Graphics Standards", IEEE CG&A, Vol. 6, No. 8,
August 1986

CAD85 CADCAM Communications,
"A Practical Guide to Exchanging CADCAM Data Using the IGES Format",
Department of Trade and Industry, 1985

DEUS84 Deusen, E.,
"Graphics Standard Handbook",
CC exchange, Laguna Beach, 1984

ENDE84 Enderle, G., K. Kansy, and G. Pfaff,
"Computer Graphics Programming",
Springer-Verlang Berlin,
Heidelberg 1984

ENDE86 Enderle, G.,
"Interfaces for Storage and Communication of Computer Graphics Information", from the book
Hopgood,F.R.A, R.J.Hubbold, and D.A. Duce,"Advaces in Computer Graphics",
Eurographics Seminars,
Springer-Verlang, Heidelberg, 1986

SELE87 Selective Update:
"Two draft standards available for public review",
IEEE CG & A, Vol.7, No. 3, Mach 1987

STRA86 Straayer, D.H.,
"Setting Standards",
Computer Graphics World,
November 1986

WILS87 Wilson, P.R.,
"A Short History of CAD Data Transfer Standards",
IEEE CG&A, Vol.7, No. 6, June 1987

A SURVEY OF MOST IMPORTANT AND OUTSTANDING METHODS FOR SOFTWARE ENGINEERING

József Györkös
Ivan Rozman
Tatjana Welzer
University of Maribor

UDK 681.3.006

With the presentation of most important methods of software engineering we want to contribute to the better knowledge, application and development tools that demand a systematic procedure for the software design. We have mentioned basic methods of requirements engineering and structured design which were already developed in the seventies. It was not earlier than in the eighties when complex tools for computer aided design of software were formed from these methods. In their essence, these methods may be devided into those which solve the problems by analysing the data flow and those which solve the problem by decomposing the data structure. The most important factor to estimate the suitability of application of a method is the level of covering the phases in the software life-cycle. The behaviour in real-time environment and the possibility to create consistent entiti-relationship models for more complex information systems are important in dependence upon the system.

Z objavo pregleda pomembnejših metod programskega inženirstva želimo prispevati k boljšemu poznavanju, uporabi in razvoju orodij, ki narekujejo sistematičen pristop k snovanju programske opreme. Nanizali smo temeljne metode inženirstva zahtev in strukturnega snovanja, ki so nastale že v sedemdesetih letih. Šele v osemdesetih letih so se iz njih izoblikovala kompleksna orodja za računalniško podprto snovanje programske opreme. V osnovi se metode delijo na tiste, ki k reševanju sistema pristopajo z analizo toka podatkov in tiste, ki to izvajajo z razgrajevanjem strukture podatkov. Najpomembnejša postavka pri ocenjevanju primernosti uporabe neke metode je stopnja pokritosti faz življenjskega cikla programske opreme. V odvisnosti od sistema pa je pomembno tudi obnašanje v 'real-time' okolju in možnost kreiranja čvrstih entitetno-relacijskih modelov za kompleksnejše informacijske sisteme.

Ø. Introduction

A multitude of actions leading to a consistent, realiable and well documented software is called **Software engineering**. In the attempt to normalize the "multitude of actions", several methods have been developed from the sixties onwards. Regarding their applicability and above all their "usableness", certain methods did not reach their climax earlier than in the eighties and most often as a base for automatic tools for software design. Let's see a brief survey of methods of software engineering /Kell87/

i) the sixties may be called also the period of crisis in the software development because it was recognized that it is impossible to create effective programs without a sistematic procedure. In this period the philosophy of **Structured programming** was formed (important authors : Bohm, Jacopini, Dijkstra).

ii) in the seventies the previously mentioned recognition caused that various technologies were developed (structured analisys, design) and term **software life-cycle** was introduced. In this period, new methods lived only in scientific circles, therefore this period is called "the period of searching for panacea" (important authors : Constantine, Myers, Yourdon, Ross, DeMarco, Sarson).

iii) in the eighties, the methods of the seventies experience a revival. With the integration of some methods a series of effective and commercially successfull **automatic tools** for software development appeared.

In the introduction let us consider the term " software life-cycle". The software life-cycle covers six basic steps /Porc83/:

- Requirements Analisys
- Functional Specification
- Design
- Implementation
- Validation
- Maintenance

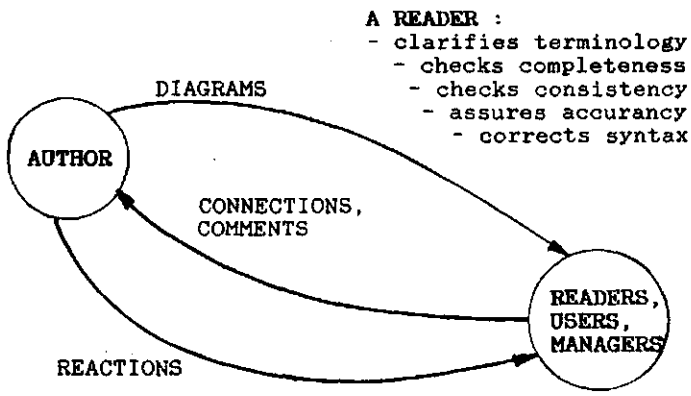
1. Requirements Engineering as a Part of Software Engineering

When we are solving problems by means of software we make a photo-copy of the real world in a logically clipped form. This form must substitute the reality in certain characteristics that are required. Therefore in the complex process of software engineering the requirements specification is of essential importance. As a rule, in this process are envolved the developer of the project and the customer. The process of requirements engineering, that is the cooperation mentioned, can be gathered into four principal points /Press87/ :

i) recognition of the problem
ii) development and synthesis
iii) specifications
iv) survey evaluation

The coordination between the developer of the project and the customer is made by the analyst (who is often called the system analyst), the system engineer, the programmer/analyst and the like. He must distinguish himself by his adaptability, ability to abstraction, communication, knowledge of the environment (hardware, software) in which the project will be realized.

Picture 1a shows the process to accord and define the requirements, (see /Ross85/). This process, called the reader/author cycle, is a part of the method which will be considered in detail later (SADT - Structured Analysis and Design Technique).



Picture 1a "Reader/author" cycle of SADT Method

Besides SADT method several other methods for software analysis and implementation of specifications are developed. This is the requirement engineering. Each method posesses a specific procedure and therefore also a different notation. The methods pass the above mentioned four points and must meet the following three demands:

a) Understable presentation of informational and functional domain in order to analyse the requirements of the problem.
Information flow shows the way of data transformation when the data pass through the system. As the result of studying the information flow the data structure for the processed system is obtained.
The **functional** requirements /Roma85/ describe the dependence of the components upon each other and upon their environment. The whole system, the program or the element of hardware can appear as a component. A conceptional model is the result of fulfilment of functional requirements. Its level of understability should be adapted to the environment for which is determined. *Non-functional* requirements cause a problem because they can exert an essential influence on the complexity of design. Many difficulties cannot be known in the early phases of design (it is difficult to determine e.g. the influence of the required level of software reliability). The software reliability is closely connected with the powerful testing tool. It is difficult to foresee the influence of the "human factor" and the respondence of the finished product to the errors. It is impossible to formalize the process of maintenence completely.

b) Division of the problem into understandable and surveyable partitions.
The division of the problem into easier and more understandable parts is carried out by a vertycal hierarchical decomposition or with a functional decomposition in the horizontal hierarchy.

c) Logical and physical presentation of the system.
The task of the requirements engineering is to clarify **what** should be realized and not **how** it should be done. Logical presentation of the problem forms the base for software system design. The analyst will not include the phisycal presentation until the logical presentation is faultless. It is necessary to define e.g. exceptions in the hardware configuration, system for database management or specificity of the applied operating system.

In the process of analysing the user's requirements a document is formed which is often called *specificatons*. A standard is existing for this kind of document (*The National Bureau of Standards, IEEE Standard No. 830-1984*), but the automated methods of software engineering do not follow it entirelly.
It can be concluded that the task of the requirements engineering is to introduce a systematic and inform notation of the information and functional analysis to solve a software system. According to the software life-cycle, mentioned in the introduction the first two points are met by the requirements engineering. The development trends in the eighties support the development and application of computer aided integrated tools for software engineering : **CASE** Computer Aided Software Engineering. They are overbuilt and the most effective among them meet the development all phases of the software life-cycle. In the following chapters, some tools will be treated in detail.

2. Methods of Software Engineering

2.1. Basic Division of Methods

The basic division is taken from /Press87/. The selected division implies to the phase of system analysis, thus it is undependent from the design method. The latter is chosen according to the obtained functional decomposition :

i) *data flow* oriented methods
ii) *data structure* oriented methods

In literature the third group is called *automated methods*. It is true that in this group the methods were made as automated ones and the philosophy of SADT method (author D.T. Ross) served as the starting-point to design methods from the first two points. Nowadays many methods from the first two mentioned groups are automated in the form of CASE tools. SADT method will be described in detail in a separate, third chapter.
Our description of methods is limited (following our opinion) to most effective and

most popular ones. The criterion of effectiveness is how many phases of a software life-cycle are covered by the method.

2.2. Data Flow Oriented Methods

All methods of this class have in common that they perform the structural analysis by means of *data flow* charts. So they give preference to the data transfer through the system over the information structure on which the system is based. The syntax of data flow charts is taken from /Gane79/ bubble chart and mainly from /DeMa79/.
Tom DeMarco describes very accurately the processing of analysed system until the functional decomposition is reached. The continuation of processing by means of structural design is taken from /Your79/. On the base of literature cited automated tools from the field of software engineering are constructed. The stress is laid on the system analysis by means of data flow charts. Nowadays the following products are availabile on the market :

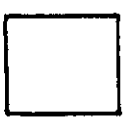
- ANALYST/DESIGNER TOOLKIT; product of the firm Yourdon Press, New York
- HP Teamwork SA/SD/RT; product of the firm Hewlett Packard
- TEK CASE; product of the firm Tektronix
- CASE; product of the firm Microtool, Berlin

SASD (Structured Analysis Structured Design) is the common name for the methods described. It is typical for SASD that it covers all phases of life cycle in the software development ("life-cycle modell" /Kell87/). SASD offers a good survey over the project and enables a team work. An effective functional decomposition from the top-down is the reason for it. As in the of this method a series of understandable tools and those being near to the man is used (a detailed description is given in the following chapters) the method permits immediate correction of the existing faults.
Automated tools always offer an up-to-date version to all members working on the project what is of special importance when individual members of groups do not work in the same place.

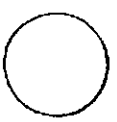
2.2.1. System Analysis of SASD
2.2.1.1. Data Flow Charts

The data flow charts can be used on every level of system abstraction. They consist of four basic building blocks (see Picture 2.2.1.1.a) from which the fundamental system model is constructed first. Then it is decomposed into several more detailed and understandable charts. The basic model is called level Ø1 of the data flow charts (further called DFC).

External entity represents the source or destination of system data. The system data can be the elements of hardware, interactive intervention of the usere or connectoin with other software systems.



Process : this sign is used when data are transformed in such a way that new data are obtained or the existing ones are converted.

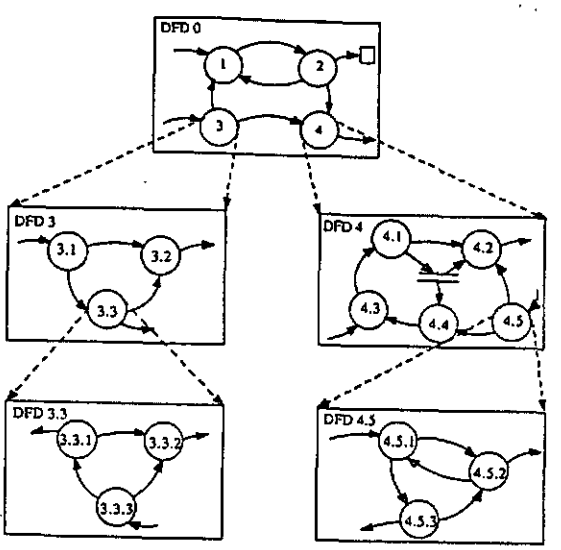


Data flow : serves to connect other basic elements of DFC. The arrow shows the direction of the flow. Each flow should possess its own uniform name.

Data storage ; the symbol means a file or an interface where the data are stored or from where the data are obtained.

Picture 2.2.1.1.a Basic elements of DFC

In order to follow the information flow best when analysis is carried out it is advisable to name the processes at the end. Data flows (information flows) that condition the necessary process form the base and not vice versa. Unfortunately, automated tools require to name the processes immediately. The picture 2.2.1.1.b shows the decomposition into the depth of an imaginary problem by means of DFC.



Picture 2.2.1.1.b Decomposition into the depth by means of DFC

2.2.1.2. Data Dictionary

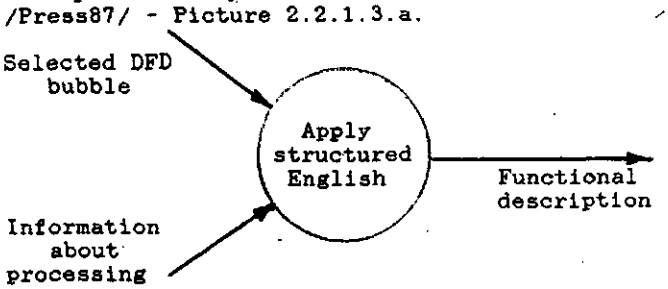
Each arrow in the data flow chart means one or more data elements of a piece of information. The data element /DEMA79/ cannot be decomposed into its components. Therefore it forms a basic element of the data flow.
In the data dictionaryevery data flow should be decomposed into elements. To do this a special notation is needed:

symbol	meaning
=	equals
+	logical and
[¦]	logical or
{ }^{n}	n iterations of bracket contens
( )	optional data
* *	comments

In the design of information systems that are aided by powerful data bases the data dictionary and data store /Gane79/ are needed to model the relational shema in a normal form (Codd's normal forms) /Show87/. The fact that the relation shema of the data base of the system designed can be formed by the structured analysis / Gane79/ essentially contributes that this method can be used. Here we find a linking point with methods that are concentrated on the data structure.
E.g. the automated tool TEK CASE alone forms the data dictionary to such an extent that the necessity to describe the undefined data flows is shown. In the data dictionary the syntax of the record is controlled automatically.

2.2.1.3. Functional Decomposition

With the data flow chart and the data dictionary it is satisfied to the information domain of the problem analysis. The transformations (processes) in the product are described as a functional domain. For this purpose structured natural language, most often " Structured English ", is used. Such clipped language is called Program Design Language - PDL.
Let's show the formulation of the functional decomposition by means the data flow chart /Press87/ - Picture 2.2.1.3.a.



Picture 2.2.1.3.a. The Process of Functional Decomposition

The dictionary of the language used for the description of processes should contain english words in the imperative form, expressions from the data dictionary and reserved words for the description of hidden logic (for this purpose the usage of decision tables is recommended). The sintax of the language used for the description of the processes permits simple sentences such as : PUT, GET, REQUEST, etc. and obligatory includes activity based constructions for the description of sequence, choice and repetitions (IF, CASE, REPEAT, WHILE, etc.).

2.2.2. System Design SASD

System design of the SASD method is taken from the source /Your79/. The transition from the data flow to the first phase of structured design, which is called structured charts, can be made in two ways. The first and most often used way is the **transform analysis** The second way, called the **transaction analysisis** because of its exaggerated formalization less often used.
The nucleous of Transform analysisis that the data flow charts are devided into the afferente part, central or transformation part and efferent part. The basic three components of the structured design are described. In the chapter 2.2.2.4. the example of the transition from the data flow charts into the structured chart is described by means of transform analysis.

The design by means of operation analysis is more effective from the transform analysis only in those cases when the centre of the transformation cannot be uniformly identified.

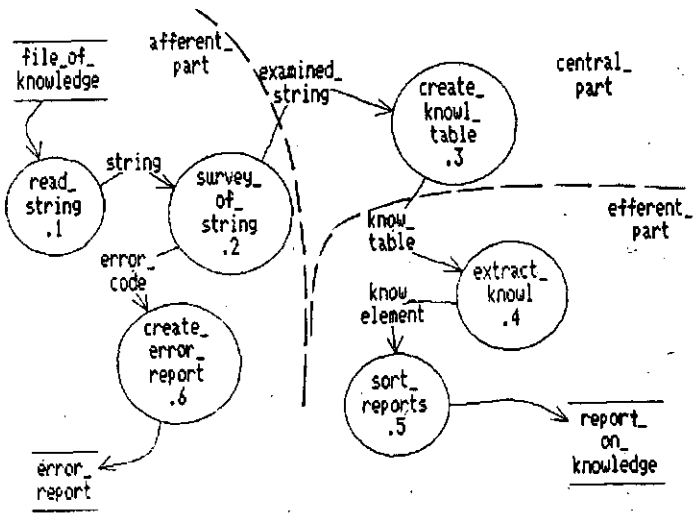
This is the case in split data flow charts (e.g. several output flows). The process that splits the input flow is called transaction centre. Around this centre a suitable structured chart is organized. Let's see the basic tree phases of structured design :

a) Structured Charts
The data flow chart shows a network of processes that needn't be connected in the final program in the same way. The model of the system in the form of software modules is ensured by structured charts. The term 'module' dendes procedures or functions in the target programming language. Thus, the structured chart directly shows the hierarchical structure of software.
More about the syntax of structured charts can be found in the literature mentioned /Your79/. The most important characteristics are described by the words :
- *rectangles* are used to denote modules. External modules, e.g. libraries are denoted by double vertical lines.
- *arrows* connect modules into various hierarchical leves.
- *arrows with circlets* denote communication interfaces between modules; the direction of the arrows shows the direction of the control paths and data items.
- *special symbols* give information on the procedurality of the system; these are iterations, conditional choices between modules, common blocks.

b) Data Dictionary
The data dictionary used in the structured design is *the same* than that used in the structured analysis. Here it is completed with new data obtained from the structured chart.

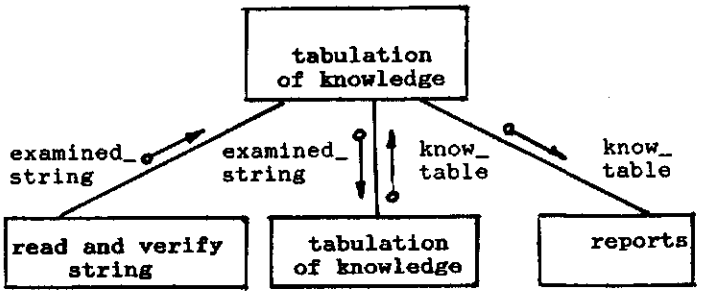
c) Description of Modules
The task of the description of modules is to explain the activity based characteristics of the modules. A pseudocode is obtained which is the direct input into the program implemented in the selected program language.

A simple example to create and follow the knowledge base is given. The picture 2.2.2.a shows the data flow chart of the problem that was devided into three basic parts, according to the *transformation* analysis.



Picture 2.2.2.a Division of the data flow chart into three parts

Each part in the data flow chart has a corresponding functional part in the tree of structured chart. The root of the tree is a new functional component which connects parts (picture 2.2.2.b). The central part of the division of data flow chart can also play this role. The starting hierarchical structure of the problem is obtained. As the processes are not described acurately by the individual functional components a further factorizing within individual divisions in the data flow charts is needed. This is done by top-down design.



Picture 2.2.2.b Structured Chart

2.2.3. Implementation and Testing in SASD

Implementation of the system is based on the top-down procedure and adding (incrementation) of new modules. Incremental procedure demands that each module should be developed and tested separately and then in combination with other modules. In such procedure the member of possible faults is decreased and the testing costs of the system are lowered.
The optimization of the implemented system should be done as late as possible and only in cases when performance tests not uncet the requirements. In this case the optimization should be concentrated on selected modules because it exerts a bad influence on important characteristics such as performance, applicability, reliability and simple maintenance of a well designed system.
In the field of testing and evaluation of software reliability the tendency is to automate the tools with uniform criteria. Additional information on practical access to this problem is found in literature /Rozm87/.

2.2.4. Application Maintenance and Owerbuilding in SASD

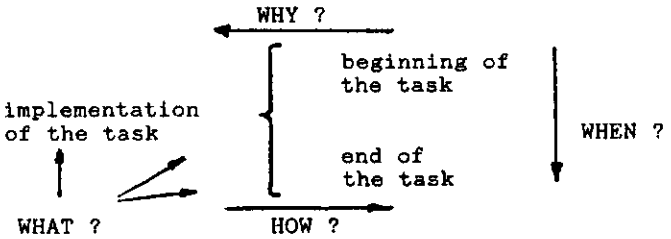
If SASD is formed through all phases into a concluded form, an exhaustive reference-book is given. The documentation on original design is given by data flow charts and structured charts, the definition and organization of data is given by data dictionary; an accurate description of processes and module structure is given, too.
Eventual corrections in testing modules should be stored as a certificate on suitability of the system and as an aid to overbuilding.
The maintenance of the system is simple. The influence of hardware changes or the modified requirements of the user can be thorougly studied in the documentation through all development phases of the system.

2.3. Data Structure Oriented Methods

As we learn from the title of this chapter these methods lay a greater stress on the construction of regular data structure than to the course of data structure. Two methods are mentioned that specifically solve the problem. They have four common points:
i) each method demands that the key information object, that is the entity, and the operators or the process should be determined at the beginning;
ii) the second common point of these methods is that their starting-point is a hierarchical structure of information;
iii) the data structure should be described by basic procedural constructs - these are the sequence, decision and repetition;
iv) these methods contain effective tools to convert the problem from the hierarchical data structure into the suitable software structure.

2.3.1. Warnier - Orr Method

From the theory of Warnier-Orr charts the DSSD method (Data Structured System Development) was developed. Warnier - Orr charts implement a hierarchical analysis of information domain. The global picture presented as multitude is decomposed. It is further decomposed into a series of submultitudes. Everithing is interconnected by basic procedual constructs. Braces are used to devide hierarchical levels. A quick and effective explanation of the meaning of the actions which are possible by Warnier-Orr notation is shown in picture 2.3.1.a /Higg79/.



Picture 2.3.1.a Directions in Warnier-Orr charts

We can see that the reading of the events from top of the chart complex downwards gives a sequence of temporal course of events. In the left direction the explanation of events is 'globalized'. To the right, there is a micro-level of described system.
The first step in the system analysis is to *define the exit of the process*. Then the logical data structure and later the corresponding physical structure are defined. In this phase the entity-relationship model can be formulated. The design of physical model follows from bottom-up but we should not think that Warnier-Orr method is based on botton-up design because the whole logical composition was performed from top down. Practical experiences of design with this method are described in literature /Gyor86/ and /Rese86/.

2.3.2. The Jackson System Development Method

The JSD (Jackson System Development) method /Jack83/ ensures a methodical way to design complex problems. The expected design result is an objective and repeatable system which is independent from the creativity of the designer. The designing process can be adjusted and is heavily influenced by the user.
The *basic idea* of this method is that it is possible to design for the user an interesting part of the real world by means entities and actions. The entities are basic elements that can be recognized by the user. Actions are implemented over the entities described and can be changed from one state to the other (the term 'states' means various phases in the life cycle of entities).
With the definition of entities and actions and with the formulation of structural shema the design phase of a model is concluded.
In the following phase a suitable distribution and definition of individual functions is made. The functions are included into the existing model at a definite moment and under definite circumstances which ensure a correct operation of the functional model.

JSD has a very rigid delimination between design and implementation. The application of the existing hardware and program languages by means of which the designed system will be realized in a classical way or by JSP (Jackson System Programming) will be not included earlier that in the last phase, that is in the phase of implementation.
JSP is also a data-oriented method. It is meant for program design and it is based on the structure of input and output data. The basic ideas of program design by means of JSP are transfered into the design of greater problems and systems. This means that JSD forms an extension of JSP. Everithing what is used in program design according to JSP can be used in system design according to JSD. The common points and differences between the two methods are shown in the following table (Picture 2.3.2.a).

action	JSP	JSD
description of the problem	terms of sequence processes	terms of sequence processes
the model consists of	structured charts	a multitude of unlinked processes for the description of temporal behaviour of entities
description of the problem is concluded by	implement.of uniform proc. structure	adding and impl. of processes which form the problem described
implementation	problem can be described immediatelly	description of the problem is converted into suitable form

Picture 2.3.2.a Comparision between JSD and JSP

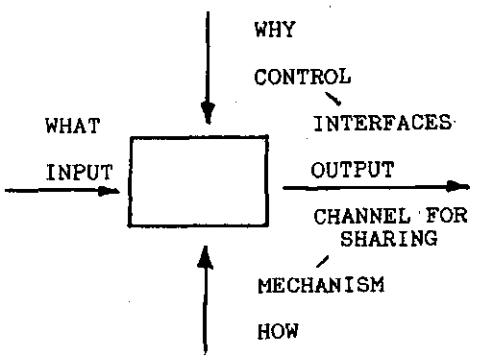
The JSD method is meant to design system in which special attention should be paid to temporal condition. JSD is used to design a wide spectrum of applications based on 'on-line' or 'batch' manner. These applications are taken from the real world and temporal extension is of supreme importance.
From the literature /Came86/ it is seen that this method is most often used in Great Britain where many systems realised by means of JSD are in operation. The method is widely used in Western Europe and less often in North America.

3. Structured Analysis and Design Technique (SADT)

This method was developed in the seventies, in the firm SofTech, Inc. when they searched for an effective tool to describe the software architecture of large systems. Douglas T. Ross /Ross77a/ and /Ross77b/ was the first who wrote on SADT. The latter source deals with the applicability of this method for extremely activity based and data-strong systems. SADT is cited as a reference for methods such as Yourdon, Jackson, Warnier-Orr, Petri nets, The multitude of PDL-s (Program Definition Language), pseudo languages and for typical data-oriented tools Codasyl, Entity Relation Attribute and others.
SADT is very effective in the early and late phases of software life-cycle. The detailed design mekes a bottle neck. SADT can overcome it with the inclusion of poverful languages for the definition of process (PDL). The most effective tool of the SADT method is treated in detail. These are 'box-and-arrow' charts which describe activity based and data aspect of the analized system. These charts are called graphic language of the structured analisys (in SADT, of course).

3.1. Graphic Language of Structured Analysis SADT

The basic guideline in the structured analysis is *understability* and *simplicity*. Therefore this method uses the decomposition from top down to the most simple basic elements. The graphic language is based on the *structured analysis boxes* (further called SAB). In the picture 3.1.a the extensiveness of such basic element is shown. The features of individual SAB are described with ICOM (input, control, output, mechanism) codes. Each SAB has its own number. On every level SAB can be decomposed into hierarchical lower components (the relationship parent - child; like in data flow charts). The decomposition cannot be done on the lowest level (atomic level) because shown system must be entirely understandable.



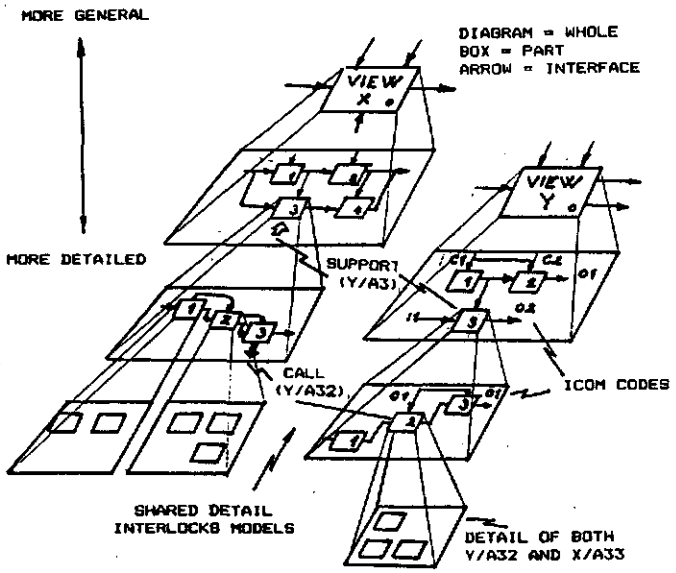
BOX IS
FORM = DATA
OR { DUAL OPPOSITES }
TRANSFORM = ACTIVITY

Picture 3.1.a The extensiveness of SAB

The *model* is called a collection of interconnected charts. the model /Ross85/ defines: M is the model of A if M can be used to answer the questions that are put about A. The quality of the model is determined by the extent of questions and suitability of answers".
Models consist of a multitude of SAB by which the decomposition of system is made. Two kinds of konnections, shown in the picture 3.1.b are known : support connections (support arrow) which are actually global nests in both models and call connections (call arrow) which can be imagined as a call of subprograms, in the terminology of program languages. The model syntax permits also that the recursion is shown.
The graphic form of requirements definition can be formalized by RML (requirements modelling language).

3.2. Applicability of SADT

Besides having influenced the development of very effective methods the SADT idea is successfully used in the development of projects dealing with various fields (formulation of requirements in various fields of multinational societies, design of complex business systems, development of telecomunication systems and unfortunately many complex military programs). It cannot be said on which field is SADT most successful. With



Picture 3.1.b Decomposition and multiple model

combined activity/data models and thorough hierarchical decomposition the method covers a vide sphere of requirements. The author of the method, D.T. Ross, aknowledges that the design and automation of design of detailed analysed system present is bottle-neck. A very accurate requirements definition and excellent documentation made the method popular to design large systems.

4. Conclusion

In this paper an informative presentation of some methods that proved themselves in the development of software systems is given. In spite of the fact that the methods like PSL/PSA (Problem Statement Language/ Problem Statement Analyser), TAGS (Technology for the Automated Generation of Systems), IORL language (Input/Output Requirements Language) and SREM (Software Requirements Engineering Methodology) are not treated we should be aware that these automated methods are widely used in the USA.

Automated methods - each automated method is based on certain formalism. The requirements for automated methods must be very strict because possible errors in early development phases are in exponential increase in the following phases. It is said that those methods are good that with incorrectly presented requirements give no final results and the designer can not be mislead. In such cases the procedures within method should give results and this effect should be increased by automation. The creativity of the designer and his ideas still represent the dominant tool. The method should only orient him correctly, warn against the faults and formulate his ideas.

Trends of CASE tools development

Besides the fact that the tool should be based on a proved formal design method which, supports the development of a program in all its phases, other requirements should be taken into account as well. These are dependent on the environment in which tool will be used.

In general, we can speak about three imortant trends in the development of CASE tools.
First we speak about the suitability of tools for the *real-time* system development. Methods deriven from data flow charts often include the real time extension. For the present state of the development of CASE tools it is tipical that too little attention is paid to the requirements of mathemeatical modelling and simulation. The necessity for immediate and detailed knowledge of hardware makes trouble the real-time systems are interrupt or event driven.
The second trend is to develop such tools that can *model the data base* by means of which the system is aided symultaneously with the development of the system.
The development trend of CASE tools aided by the *artificial intelligence* is gaining in importance. Expert systems can be used to control syntactical and mainly semantical errors of automated tools. In order to fulfill the communication between the designer and the tool as much as possible object - oriented methods are developed. They enable a natural connection between the designed model and the reality described by the model /Borg85/.
An interesting problem arises when we try to design unstructured AI software systems by conventional methods of software engineering. Ideas to solve this problem are found in literature /Part86/. The author warns us that for the present it is impossible to offer an accurate and formalized method to design AI software systems.

LITERATURE :

/Borg85/ A.Borgida,S.J.Greenspan, J.Mylopoulos, "Knowledge Representation as the Basis for Requirements Specification", IEEE Computer, april 1985

/Came86/ J.R. Cameron, "An Overview of JSD", IEEE Software Engineering, 1986/2

/DeMa79/ T. DeMarco, "Structured Analysis and System Specification", Prentice Hall, N.J., 1978

/Gane79/ C.Gane, T.Sarson, " Structured System Analisys ", Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979

/Gyor86/ J.Gyorkos,T.Dogsa,I.Rozman," Warnier-Orr Diagram Application Experiences", 8th International Symposium Computer At The University, Cavtat, 1986

/Higg79/ D. A. Higgins, " Program Design and Construction",Prentice-Hall,Inc., New Jersey, 1979

/Jack83/ M.A. Jackson, "System Development", Prentice Hall International, 1983

/Kell87/ G.Kellner (CERN Geneva), Papers from " CASE Seminar Tektronix", Wien, 1987

/Part86/ D.Partridge, " Arifical Intelligence Applications in the future of software engineering", Patridge/Ellis Horwood Ltd., Chichester, 1986

/Porc83/ M.Porcella, P.Freeman,"Ada Methodology Questionnaire Summary", ACM Software Engineering Notes Vol 8 No 1 Jan 1983

/Pres87/ Roger S.Pressman,"Software Engineering ", McGraw-Hill Book Company, New York, 1987

/Rese85/ I.Rozman and others: Research report on Informatyon systems and Artificial Intelligence, C2-0522/796-85, Faculty of Technical Sciences Maribor, 1986

/Roma85/ G.C. Roman, " A Taxonomy of Current Issues in Requirements Engineering ", IEEE Computer, April 1985

/Ross77a/D.T. Ross, "Structured Analysis: A Language for Communicating Ideas",IEEE Trans.Software Eng.,Vol SE-3, Jan 1977

/Ross85/ D.T.Ross,"Applications and Extensions of SADT ", IEEE Computer, April 1985

/Rozm87/ I.Rozman, T.Dogsa,"Empirical software Reliability Models", The 2nd Beijing International Symposium On Computerized Information Retrieval, Beijing China, 12/1987

/Shov87/ P. Shoval, M.Even-Chaime, "Data base shema design: An experimental comparision between normalization and information Analisys", DATA BASE, vol. 18, 1987

UDK 681.3.068

Anton P. Železnikar
Iskra Delta

Characteristic problems of mathematical insufficiency and inadequacy appear in the rational, especially mathematical expression of information, i.e., in the expression of processes of informational arising. The mathematical formalism, which was quite sufficient for the purposes of describing some well-determined linguistic constructions, does not satisfy in its traditional form, when the reality, which is arisingly changing, unforeseeable, and developing, has to be expressed. Because of such (dynamic) informational phenomenology, a critical view in the groundwork itself, which constitutes the classical symbolic or formal logic, is necessary. Besides, it is necessary to omit the rigid and for today's circumstances of thinking too hard and unacceptable mathematical definitions, as for instance, the logic itself, relation, set, function, differential, integral, category, etc. All this does not mean that mathematics is losing its previous role, but it becomes evident that mathematics can no longer seize constructively (substantially, completely, or credibly) and scientific-developmentally into new realms of cognitive development.

The contribution of this article certainly remains essentially far from a new, more adequate formalization as is the classically mathematical one. However, many problems are much better enlightened. The mathematical logic, proceeding from the thinking of the previous century and from the first half of this century, is historical-developmentally useful. However, its groundwork has to be reformulated in such a manner which will enable an easy derivation of various general and concrete logics. Within this concept, the contemporary formal logic of mathematics and mathematical systems became only particular cases and, in their essence, conceptually too reduced. From the context of this article it is evident how particular logical, relational, set-theoretic, functional, differential, integral, categorial cases of notions are bounded (limited) and inadequate and also, how their dynamic reformulation would be possible. But, this article represents only a soft introduction into this new discourse and an investigation of particular mathematical inadequacies. The goal of this preliminary discourse is, of course, to lay out the way which leads to something termed informational logic. In the framework of such new logic, it will become possible to develop essentially changed concepts of set, relation, function, category, etc. as was the case in today's and yesterday's mathematics.

The context of this article is certainly proceeding into the direction of a general criticism of those kinds of rational understandings that in the most cases deteriorate into the rigid rationalism and that, like traditionalism, set essential blockades against the progressive thinking. Thus, also of these kinds of rationality it is true that after a period of development they degenerate into their counter-information which is nothing more than irrationalism and, lastly, an unacceptable dogmaticism for the progressive thinking.

0. Introduction

In the beginning there was information. *The word came later. The transition was achieved by the* development of organisms *with the capacity for selectively exploiting information in order to* survive *and* perpetuate *their kind.*

Fred I. Dretske [1] vii

Is a rational understanding versus a rationalistic one in the form of a loose definition of information in a broader sense (as discussed in [2, 3, 4, 5]) at all possible? What are the possibilities of developing a general (not only the classic, information-theoretical) concept of information in a quasi-mathematical (mathematically non-traditional) way (e. g. through the informational formalism of symbolic logic, set, function, differential, derivation, integral theory, algebra, geometry, theory of categories, etc.)? What does the

classical (communicational, stochastic) theory of information represent today [6]?

It is evident that before a further discussion concerning a rational or even rationalistic view of information is possible, the mathematics-characteristic way to a rational (mathemational instead of mathematical) formalization (abstraction and the corresponding symbolism) of information has to be enlightened. Aready simple examples can show why the classic rationalistic (mathematical) way is abysmal and how the deficiency of the discussed essential, rationalistically valid concepts is coming to the surface.

1. Informational Models

> *Philosophers ... still seem disposed to think about knowledge, perception, memory, and intelligence with a* completely different set of analytical tools: *evidence, reasons, justification, belief, certainty, and inference. There is, consequently, a serious communication problem.*
>
> Fred I. Dretske [1] viii

Several simple and sophisticated models dealing with the rationalistic formalization of informational subjects and objects can be distinguished. Scientifically, the most direct rationalistic informational model is the so-called theory of information. Its concepts are narrowed into mathematical measures of information and can be numerically "calculated" by different mathematical functionals. The theory of information constitutes the so-called naive (rationalistically reductional, comprehentionally simplistic, informationally quantitative) model of information and represents merely (exclusively) the field of relevance of the so-called generalized theory of (technological) communication systems.

The naive model of information can be advanced already by redefining the components of the generalized communication system in a more complex way, e.g., by raising the question of the nature (the essence) of information source (arising of information or coming of information into existence), information channel (propagation of information through biologically modulated and filtered paths), information coding (various phenomenological transformations of information), information (or merely data) transmission and reception in living and artificial environments, etc. On this way, examples of living information models (philosophical investigations) can be helpful in the development of a critically much more adequate (informationally non-naive) discourse concerning information, than that following from the classically narrowed information theory. Last but not least, a general model of information can be imagined, embracing some essentials of already existing rational, rationalistic and intuitive models of information.

2. The Contemporary Theory of Information

> Communication theory *purports to tell us something about information, if not what it is, at least how much of it there is. ... this is controversial. ...* The mathematical theory of information *may be an elegant device for codifying the statistical features of, and mutual dependencies between, those physical events on which communication depends, but information has to do, not with vehicles we use to communicate, but with what we communicate by means of them. A genuine theory of information would be a* theory about the content *of our messages, not a theory about the form in which this content is embodied.*
>
> Fred I. Dretske [1] 40

Information as a notion has different meanings in different environments (for instance, in different lingual cultures and in different scientific disciplines). Information does not have the same meaning in the realm of common sense, communication theory and engineering (technical sense), crime investigation, theory of information (mathematics), information technology (computer science), information science (linguistics, information retrieval), or even in a general information philosophy (information as information).

Today's theory of information or information theory, which is founded firmly on mathematical, telecommunicational, and computer methodologies (and technologies), is rationalistically traditional and comprehends, for instance, the following applicative and theoretic areas:

- application of information theory,
- complexity of information,
- communication systems,
- cryptography and security,
- data networks,
- detection and estimation,
- distributed information processing,
- error-correcting coding,
- multi-user information theory,
- pattern recognition,
- processing of audio and visual signals,
- *Shannon's* theory,
- source coding,
- stochastic processes, etc.

Today's theory of information is grounded on merely some characteristic mathematical and technical concepts, but in spite of this fact, it has created many useful concepts by itself as a specific discipline. It must be stressed that information theory, in principle, ignores the problems of informational arising (e.g. coming of information into existence, irrespective of the nature of informational source).

Within the theory of information the so-called information theorist's definition of information is used in its mathematical (or engineering-technical) sense. This concept has the name information which is (narrowed) enough for the purposes of the theory of information.

However, it does not embrace a common idea of information appearing within different living verbal languages or even living organisms.

Information can be defined in mathematical terms in several ways. The amount or quantity of information is a measure of time or cost of transmitting messages from the information source to the destination. The information source is extremely restricted to be capable of putting forth *n* probable messages, so that information can be defined as log *n* and generalized to the entropy function. Information capacity of a channel (the number of distinguishable signals in a time unit) gives a measure of how long it takes to transmit the message generated by source.

An information source has only the role of information supplying and, in fact, it is not concerned with information arising or with the fundamental problem of how and why information is coming into existence and what information represents when it arises as information in general. The information source simply delivers codified information where the arisen information was coded by a source coding process, so it can be fed into the channel. Decoding is therefore needed to transfer information from the channel to the receiver. Sometimes, because of noise in the communication system, the information obtained by the receiver is erroneous. This is the concept of the noisy channel.

In information theory, one of the most important fields of mathematical science, the amount of information is estimated by means of entropy. Several characterizations of entropy have been given by *Shannon*. Information theory combines various methods of probability, statistics, functional analysis, *Fourier* analysis, and algebra.

3. Logic qua Information

> *Logic is the science of the pure Idea; pure, that is, because the Idea is in the* abstract medium *of Thought. ... Logic might have been defined as the* science of thought, *and of its laws and characteristic forms. ... If we identify the Idea with thought, thought must not be taken in the sense of a method or form, but in the sense of the* self-developing totality *of its laws and peculiar terms.*
>
> G. W. F. Hegel [9] 25

3/0

Logic is close to a doctrine of essence, i.e., of the absolute. This doctrine is purely informational and arises, for instance, as common sense, belief, reasoning, awareness, and like that in living cortices. Similar as language, logic as philosophical, scientific, or mathematical discipline concerns particular informational domain dealing with essence as a measure of absolute truth in its own realm. In this respect, logic belongs in the class of philosophical, scientific, or mathematical informationism.

In its foundation, logic (the Greek 'logos' has the meaning of speech, word, mind, thought) is a philosophical discipline dealing with valid (true, correct) forms of thought and with methods of scientific cognition. In this sense, logic is the study of the structure and principles of reasoning or of sound argument, with the aim of establishing the truth of propositions in the form of deductive and inductive inference. There exist several other kinds of logical inference as studies of reasoning which may be termed the deontic logic, logic of norms or logic of imperatives, logic of relevance and necessity, modal logic, logic of language, logic of believing, knowing and inferring, logic of inconsistency, epistemic logic, and, last but not least, informational logic. However, in its narrower sense, logic is the study of the principles of deductive inference, or of methods of proof or demonstration. Informational logic is the study of informational principles including also information and Informing as methods of informational inference through the coming of information into existence.

Informationally, logic is merely one of the possible doctrines of reasoning which is termed to be *rational*. The science of deductive logic has its roots in the conception of establishing propositions by means of such arguments that it would be *irrational* to reject their conclusions, having accepted their premises. The nature of logic as information will depend on the natures of truth, knowledge, and cognitive abilities of a being, and informationally, on the being's total information (a being's metaphysics). However, imperatively, logic as a cultural and as a populational information can (or better must) remain characteristically rational, i.e., culturally and populationally doctrinal (rationalistic).

3/1

The creator of European logic was *Aristotle* and his successors, the philosophers of *Megarian-Stoic* school, who have established the principles of deductive logic. At the beginning of the 17th century, *F. Bacon* introduced the idea of a new, inductive logic. The predecessor of the so-called symbolic logic was *Leibniz*, but its most distinguished creators were *G. Boole, G. Frege, B. Russel, R. Carnap, A. Tarski,* etc. In his *Begriffschrift* (1879), *Frege* introduced quantifiers and the treatment of concepts by analogy with mathematical functions. This enabled him to unify the logic of propositions (propositional calculus) with the study of those logical relationships which had previously been treated in the theory of syllogism. *Frege's* approach was adopted by *Russel* and *Whitehead* in the writing of *Principia Mathematica*, where the focus of attention was shifted firmly away from terms to propositions and their relations.

Even prior to *Fregean* innovations, the study of logic had become increasingly mathematical. The

question is how logic can become an informational discipline. Mathematics and mathematical logic are merely specific forms of information. However, one would like to construct a system of logic which would adequately embrace the realm of information. Within mathematical logic it is possible to treat a formal logical system as just another system of algebra giving rise to an algebraic structure that can be studied using mathematical methods. In the following sections we shall try to reveal some "logical" problems of informational algebraization of logic.

Logic, especially the mathematical one, is a logical construction. It uses its own language, which is narrowed into the rationalistic domain of deductive and inductive inference. Mathematical logic is a language which is formalized by specific symbolism as a form of gibberish of symbolic logic.

The effective logic uses junctors and quantifiers (logical connectives or logical operators) to construct propositional formulae (logical expressions). These formulae use logical variables and logical constants (logical operands). Junctors and quantifiers are symbols representing certain devices, independent words, prefixes, suffixes, and inflections in any language that enable one to discern the grammatical structure of a sentence.

In order to represent the logical structure of a sentence, junctors and quantifiers are introduced in some places of the sentence. The logical symbols thus have a precise (constant) function. The most commonly employed operators are negation, conjunction, disjunction, implication, the existential and universal quantifiers, and the true and false symbols. A logical proposition (formula, expression) is either true or false. In this way, a two-valued logic is constituted. Certainly, it is possible to define also a multi-valued logic using adequate logical operators.

3/2

Let us consider the universal quantifier *(x)* with the meaning 'for all *x*'. Informationally, this quantifier is questionable, for it can exist (or arise) 'at best one case of *x*'. In fact, this quantifier is pretending that all x of some domain (set, class, category) have a common property or, in other words, are equal concerning this property. In this manner, the universal quantifier hides a kind of informational equalization. Thus, we can have the following reasonable informational transformation:

for_all *x* --> at_best_one_case_of *x*

In this sense, the logical quantifier *(x)* passes into the informational one with the meaning at_best_one_case_of, which could be denoted by E? (E! is already used for 'there exists exactly one'). If we understand that the quantifier for_all *x* has in fact the meaning for_all *x* which_have_a_common_property, then *(x)* as a quantifier is informationally acceptable too.

The logical quantifier 'there exists at least one' is in the strict informational sense non-logical too. That which exists informationally is continuously changing, coming into existence. Thus, the informational transformation

there_exists_at_least_one *x* --> there_arises *x*

is reasonable. However, information can arise merely if its arising (Informing) already exists (otherwise, *x* cannot arise).

(x) has the meaning 'for all *x* which exist'. As pointed out, this quantifier could be informationally introduced for all those cases whose arising is in the domain of a generative set (a generative set *G* is defined by its characteristic informational function). For this purpose, the denotation A^{Δ} *G* with the meaning 'for all which arise within the generative set *G*', would be appropriate.

Similarly, the existential quantifier 'there exists at least one *x*' could have the meaning 'for those *x* which already arise' (in a particular case, they can exist as unchangeable information).

In metamathematics, the existential (there exists at least one) and the universal logical quantifier (for all) are bound to a fixed existence. However, in information, this bindingness concerns the coming into existence. In the informational sense, these quantifiers could be used as 'at least one *x* is coming into existence' (instead of 'there exists at least one *x*') and as 'for all those *x* which are coming into existence' (instead of 'for all *x*'). Evidently, the existential and universal logical quantifiers are losing their general meaning in the informational realm, for they represent cases of informational constancy (statics, datingness).

3/3

At this point, we are coming to the question whether it is at all possible to introduce reasonable informational quantifiers. Certainly, it is possible to say that this and that information is in the process of arising. Simultaneously, it means that the existence of information, its development, its uniqueness (two equal cases of information do not exist) and, in the future, also its vanishing or its transformation have reached such a degree of change that the identity to its previous identity is more or less not recognizable. As already mentioned, for the arising of information *i*, the quantifier A in the form

A*i*

can be introduced. This means that information *i* is in the process of arising. If *n* cases of information arise, there could be written

A $i_1\ i_2 \ldots i_n$

For instance, for all informational cases belonging to a generative set *G*, one can write

A*i*(*i* belongs_to *G*)

In a sense the arising quantifier A by itself points to the generative nature of set *G* (informational consistency between A and *G*).

3/4

The next possible question concerns the negation of information. Informational negation is, of course, not logically (in the sense of mathematical or formal logic) exclusive in respect to the observed informational original. This means that information *i* and its negation ~*i* (the sign ~ is used to denote the negation) do not represent to each other such a contrast (controversy) that, in the sense of classical logic, it would hold

i OR ~*i* = *true* and
i AND ~*i* = *false*

In informational logic, *i* and ~*i* can exist simultaneously, although the essential question arises: what is ~*i* in respect to *i*.

In the case of a simple predicative (affirmative) information (is-sentence), the answer to the last question is still possible, for instance, 'Peter is a wolf' and 'Peter is not a wolf'. But, in a much more interwoven information, the notion of logical negation is loosing or losing its sense. One can only state the fact that information and its negation are two (spacial, temporal, logical, informational) cases of information which, in general, are different.

In no case, informational negation can be an exact negation of the informational original in the classic logical sense. Negation as information is arising in the sense of informational arising similarly as the informational original. When i_2 arises from i_1, one can write

$$i_2 = Ai_1;$$

However, the quantifier A which simultaneously has the property of operation of arising, is arising by itself. And, because A is information by itself, it becomes

AA

as arising of the quantifier A. This leads to a form of the chain of arising, for instance,

$$i_n = Ai_{n-1} = AAi_{n-2} = \ldots = A^{n-1}i_1;$$

This informational expression represents a recurrent autoinformational and alloinformational operation.

3/5

The most common logical operations are disjunction and conjunction or, in the set theory, union and intersection. In an informational realm, it is quite easy to imagine the so-called union of informational cases; however, the informational intersection is faced with the problem of searching equal pieces of information in different informational cases. The informational intersection is based on equalization of informational pieces as they appear in different information.

In cases of logical quantifiers and logical operations, it is possible to observe how the rationalistic symbolism is losing its logical steadiness. Maybe that rationalism as a form of modern cognition and understanding in philosophy and also in technology became inadequately rigid, untrue, and nonsensical (in itself non-rational) to the degree where it cannot stay any longer in the exclusive position as a prevailing principle of scientific and technological progressing. The searching of principles standing outside of rationalism is certainly on the way of surpassing those doctrines which block the progressing and more critical and differently organized thinking and also scientific and technological constructing.

3/6

Besides logical operators, the so-called modal logics use various *modal* operators. These operators are constant or, in a limited form, variable statements (propositions). For instance, for *'a* believes *p'* one cane introduce *a* L *p*. However, for 'agent *a* believes *p'* one can choose L_a *p*. When a belief after some time becomes the norm, it can be set K_a *p* with the meaning *'agent a knows p'*. In this way it is possible to construct languages of logic and give semantics to them [11].

4. Relation as Information

4/0

The Latin *relatio* (-onis, f) means report, proposition, motion; throwing back; returning; repetition. Today, relation has the meaning of the act of telling or recounting, an aspect of quality that connects things, reference, etc. Obviously, relation is a *characteristic form of information*. In mathematics, relation is a propositional function of two or more arguments defining, for instance, exactly those elements belonging to a set. These elements satisfy the propositional function of the relation, for instance, in concern to the components appearing as elements of the set.

4/1

Mathematics has developed several notions of relation. One of the most general relations is, for instance, the relation *to be the element of a set*. This relation seems to be mathematically one of the most natural. In general,

mathematical relations can be determined by means of other relations, operations, existentials, quantifiers, axioms, predicates, and/or verbal, abstract, intuitive determinations, etc.

The notion, which concerns relation, is the so-called relational set. A relational set R consists of elements which are pairs (a, b) of some elements a and b belonging to some sets. Now, the relation R which is represented by the relational set R, can be understood as operation R over elements a belonging to a set, in the way that it produces elements b belonging to a set. The expression

$$a\ R\ b$$

has the meaning that b will be produced by R from a.

4/2

What is the relevance of the relation in an informational realm? The most obvious informational relation is, for instance, that information, in one or another case (place, time, process, phenomenon), is *informationally different.* If information is taken as a unity or an autonomous entity, then informational inequality between one and another informational case is the most natural relation. In the course of human social development, as controversy, opposition, antithesis, counter-information became usual, evident, and consciously regular ways of thinking, the controversy of informational difference (as the most evident relation in an informational realm) arose as an informational principle (as the counter-possibility), too. This way of inference led to the notion and *relation of informational equality.* Although informational phenomena are naturally different, they can be abstractly counter-informed as equal. A characteristic similarity of observed objects ceased as an abstract way of inference into their (informationally dazzled) equality.

The relation of equality (in the mathematical sense) cannot exist between two informational cases (informational forms and informational processes). The relation of equality belongs obviously to the most abstract (unbelievable) cases of the real world. However, on the other side, the relation of equality is the foundation of each, quite arbitrary symbolism. *Symbolic equality* qua symbolism is in the domain of informationalism (irrationalism, ideology, state-of-mind) [3]. This informational distinction for the case of the equality relation is essential, because also the so-called *sign of equality* '=' will be used in further explication of the mathematical or formal informational (mathemational) background or field of relevance. In all our cases, the sign of equality will represent only a formal, abstract (theoretical) symbolic equality.

4/3

The concept of the mathematical relational set and its understanding as operation can be conceptually transposed into the formalism and abstract understanding of information. Informing as a process of information can be comprehended as relation, i.e., as the most general informational operation. As pointed out, this operation (relational information) is called Informing.

Let us examine the most general case of the relation we call Informing I. In general, there is

$$i\ I\ i$$

This reflexive relational expression has the meaning that Informing I produces information i from information i. A more constructive relational expression is, for instance,

$$i\ I\ i_1$$

where i_1 arose by I from i and includes the so-called counter-information c.

As i changes and arises, so does I. In this way it is meaningful to set the relational expression

$$I\ i\ I_1$$

In many scientific (mathematical) and technological cases I is a constant (unchangeable, non-arising) information. Under these circumstances it is possible to create the so-called deductional chain (transitivity of relation I) of the form

$$i\ I\ i_1\ I\ i_2\ I\ i_3\ I\ i_4\ \ldots$$

or, expressed in a shorter form,

$$i\ I^*\ i_4\ \ldots$$

In the case where informing I is arising, the deduction schemes may become extremely complex and informationally interwoven, for instance,

$$\begin{array}{l} i\ i\ I \\ i\ I\ i_1\ I_1\ i_2\ I_2\ i_3\ I_3\ i_4\ \ldots \\ I\ i_1\ I_1\ i_2\ I_2\ i_3\ I_3\ \ldots \end{array}$$

In the first line (chain) we have a kind of mathematical (definitional) inconsistency, for (i, I) appears as an element of relation i (i generates Informing I from i). In the second chain, relations are Informings I, I_1, I_2, I_3, etc.; however, in the third chain, relations are i_1, i_2, i_3, etc., and both chains are mutually dependent (developing). The second chain generates cases of information i_1, i_2, i_3, i_4, etc. by Informings I, I_1, I_2, I_3, etc. where the third chain generates Informings by cases of information generated by the second chain by these Informings.

5. Sets as Information

... semantics lies outside the scope of the mathematical theory of information.

Fred I. Dretske [1] 42

5/0

The set as a mathematical notion arose by the work of *G. Cantor.* A set is a collection of objects of thought or intuition with a certain realm, taken as a whole. However, this is not a strict definition of the notion of a set. Each object in the collection is called an element or member of the set. The point of the set theory is not so much to explore the nature of a set as relations among elements of a set and the set as a whole.

Cantor's naive set concept leads to various *logical paradoxes.* His set theory was reconstructed as the *axiomatic set theory,* which is considered to be free from paradoxes.

5/1

Mathematical sets can be defined in several ways. If possible, elements of a set can be simply listed down. Finite and infinite sets can be defined by means of the so-called characteristic (or definitional) functions, which are logical predicates for elements of a set. A set is a definitional statement which determines the relation of belonging between elements of the set and the set. In this way, a set is merely static (or unchangeable) information of a collection of set's elements. The set-definitional predicate is a mathematical-logical statement, and this statement is definitionally constant (stable, unchangeable). For a set definition, the main problem is how to construct the adequate characteristic function to obtain the desired collection of elements.

In the set-theoretical discourse, the definition of the set as information is basically very important, too. If a set consists of elements representing informational entities, these elements can hardly be informationally completely independent of each other. It means that besides this set several relational sets can exist, determining the informational connectives among set elements. On the other hand, sets as informational entities in their reality can only be variable and not defined once for all. This reality claims for the possibility to define a set as a definitionally variable entity. In this case, the logical predicate representing the so-called characteristic function of the given set has to be replaced by an informational function for which predicative variability is not exceptional, but (informationally) regular.

If the so-called informational set claims for its variable definition, the set-definitional predicate or set-characteristic function must be a definitionally variable (non-mathematical or quasi-mathematical) predicate or function. In the next section we shall show some problems of defining informational functions.

Let us now introduce the concept of the so-called generative set which, in a slightly modified form, was proposed by the author already in 1975 [7]. A generative or, now, informational set *(i-set* for short) of informational elements *(i-elements)* performs as an information generator, producing information which represents elements of this set. However, this set generator or *i-set* is not a generator in the usual sense, for generated elements are not simply accumulated, but can exist as generated, they can vanish, or can be changed through their life time. Formally, the definition of an informational set can be written as

$$i\text{-}set = \{i\text{-}element \mid i\text{-}function(i\text{-}element)\}$$

In this formal expression *i-function(i-element)* represents a function *i(i)* where *i* stands for information. We shall discuss informational functions more in detail in the next chapter.

5/2

Ideology, as treated in [4], is a complex example of informational function (informational form and/or informational process). An informational function can be used to define objects (elements) belonging to a generative set. Thus, it is possible to choose a proper, subideological function for creating sets concerning ideological matters.

Let us look at an example of a generative set and the dependence of its structure on information constituting its characteristic function (informational set-definitional predicate). Let be given an informational pool of people belonging to a particular population. Obviously, this pool is a generative set.

Let us create an ideological subset of people whose behavior or suspicion of such behavior satisfies the characteristic function 'Human, who is class enemy'. It is more or less clear that in the course of time, the characteristic function of the ideological subset will depend on informational circumstances (complex informational environment), arising also in an unforeseeable way. These circumstances will contribute informationally to the structure of the subset characterized as 'Human, who is a class enemy'. It is possible to imagine how this characteristic function of the subset is arising, changing, and, under unforeseeable circumstances, vanishing or, in the course of change, disappearing. In this respect, as one can understand from the previous example, sets concerning any reality, even the most abstractly (ideologically) understood ones, in principle are generative. They are only particular cases of information or, comprehending their substantiality, are informational (ideological, scientific, mathematical, etc.).

5/3

The concept in the informational fuzzy set theory would be that an element has the information of membership in a fuzzy set as a generative set. The form of the information of membership can be in the most simple and logically reductional case that the information of membership is a probability or a mathematical measure belonging to an interval

of real values. But, the information of a fuzzy set membership can be any type of information, Informing, counter-information, embedding, etc. In this case, the information of membership becomes a kind of dynamic potentiality.

A fuzzy set has to do with the so-called *universe of discourse* [10]. A *fuzzy subset of a* universe of discourse *U* is a function

$$f: U \longrightarrow [0, 1]$$

On the sets of all fuzzy subsets defined on the universe *U*, operations of union, intersection, and inclusion can be defined, using the informational (also mathematical) predicates of maximum, minimum, and 'lower or equal'.

The concept of the fuzzy set shows how the membership of a set element depends on further information concerning this element. In this respect, the notion of the fuzzy set is nothing else than a discussion on the membership of an element. In this discussion, the most important fact is not so much the fuzzy set itself by its elements as it is the so-called membership set, which carries information and, within this information, the possibility or even potentiality for an element to be the element of a fuzzy set. In this way, the concept of the fuzzy set is reduced to the problem of setting information concerning the membership of set elements. In a way, the membership set of the fuzzy set is a set of characteristic functions approving harder or softer belonging of an element to the fuzzy set. As an arbitrary set, through its characteristic function, also the fuzzy set becomes more and more not the set itself, but its characteristic function. In this course of the discourse, the set concept becomes a set-general information, which could constitute the set as a concrete set realized as a collection of all possible elements belonging to a set.

6. The Notion of Information qua a Quasi-Mathematical Function

> *... Communication theory does not tell us what information is. It ignores questions having to do with the content of signals, what specific information they carry, in order to describe how much information they carry.*
>
> Fred I. Dretske [1] 41

6/0

The term *function* was introduced into mathematics by *Leibniz* (in 1694) to refer to certain line segments whose length depend on lines related to curves. The modern functional notation *f(x)* had been used in 1734 by *Clairaut* and by *Euler*, who defined functions as analytical formulas constructed form variables and constants. *Cauchy* introduced the terms of *independent* and *dependent* variables (1821). *Dirichlet* (1837) generalized a function as a correspondence in which the values of one variable determine the values of another (completely arbitrary functions).

Today, a function has the meaning of *mapping* or univalent and also multivalued correspondence. A letter *x*, for which a name of an element of a set *X* can be substituted, is called a variable, and *X is called the domain of the variable*. An element of the domain of a variable *x* is called a value of *x*. A letter which stands for a particular element is called a constant.

6/1

The arising of information calls for a concept of the function which as the definition of a function is *variable* (arising-like). First, let us try to define information by the so-called definitionally static function, i.e., by a sort of quasi-mathematical functional concept. In this case, a function is meant to be a mapping of a set of informational values of the functional argument *x* into (or onto) a set of informational functional values *(y)*. In principle and in general, information can be formally (approximatively) conceptualized as a *circular* system of two metainformational equations:

$$y = y(x) \quad \text{and}$$
$$x = x(y)$$

This system is the *metainformational* expression of the (approximative) nature (information) of information. Within these equations, *x* has the following meaning:

- in the first equation, *x* is the *original* (starting) information;
- in the second equation, *x* (left from '=') is the *counter-information;* and
- in the second equation, *x* (right from '=') is the so-called *embedding* (embedding Informing, embedding counter-Informing) of the counter-information *y* into the original information *x* (left from '=').

The meaning of *y* is as follows:

- in the first equation, *y* (left from '=') is the *counter-information,* arising from the original information *x* through the so-called Informing (counter-Informing) *y* (right from '=') and
- in the second equation, *y* is the *counter-information,* which will be embedded into the original information x (left from '=') through Informing (counter-Informing) *x* (right from '=').

This system of equations is questionable in many respects and some of them will be illuminated in the following discussion.

6/2

Through symbolic playing with mutual substitutions of equations, one can obtain, for instance,

$y = y(x(y(x(\dots (y) \dots))))$ and
$x = x(y(x(y(\dots (x) \dots))))$

This system represents the coming of information into existence through several *cycles of Informing*, where each cycle is understood to be a composition of counter-Informing and the embedding of information. The later system of functional equations is not very clear and the functional structuring of information, Informing, counter-information, and embedding remains hidden, in general. So, a slightly different symbolism can be introduced, where the processes of informational generation and embedding are marked by capitals *(X, Y)*. These capitals represent functional "prescriptions". Instead of the later system, there are

$y = Y(X(Y(X(\dots (y) \dots))))$ and
$x = X(Y(X(Y(\dots (x) \dots))))$

This quasi-mathematical system does not show explicitly the coupling occurring between the consequent prescriptions X and Y when the arising-embedding cycles are taking place. In this metainformational representation, the arising of X and Y cannot be explicitly identified, even this arising is implicitly present in the last metamodel. This model can be analyzed (indexed) in the following way:

$$\begin{aligned} &y_0 = Y_0(x_0), \quad x_1 = X_0(y_0);\\ &y_1 = Y_1(x_1), \quad x_2 = x_1(y_1);\\ &\dots\dots\dots\dots\dots\dots\\ &y_n = Y_n(x_n), \quad x_{n+1} = X_n(y_n); \end{aligned}$$

where in the consequent lines the so-called cycles X_k and Y_{k-1} are listed, representing counter-Informing and embedding and where i belongs to the discrete interval $[1, n+1]$. This case can be transformed into a clearer form

$$\begin{aligned} &y_n = Y_n(X_{n-1}(Y_{n-1}(X_{n-2}(\dots (y_0) \dots))));\\ &x_{n+1} = X_n(Y_n\ (X_{n-1}(Y_{n-1}(\dots (x_0) \dots)))) \end{aligned}$$

At least, it is evident that the consequent X_k and Y_k vary from one i-cycle to another.

6/3

A function $y = y(x)$ is definitionally constant if its functional prescription (its definition) is given once for all, i.e., if it does not change during its existence. It was pointed out that information needs a concept of variable functional prescription. Because of this, a prescription in the form

$$y(x) = \begin{cases} y_1(x) & \text{for} \quad a_0 <= x < a_1,\\ y_2(x) & \text{for} \quad a_1 <= x < a_2,\\ \dots\dots\dots\dots\dots\\ y_n(x) & \text{for} \quad a_{n-1} <= x < a_n, \end{cases}$$

where $[a_0, a_n]$ is an informational interval, is not adequate. This type of function represents merely a sequence of constant functional prescriptions $y_1, \dots, y_n$.

6/4

Let i be *information* (for instance, a being's total information) when in the framework of this information, *Informing* I is coming into existence. Informing I is a generative processing product of information i. A trivial description (formula) of this generation would be $I = i(i)$. This formula informs that information i has generated Informing I from (or on the basis of) information i. In this case, information i appears as the generator of I (informational host of Informing I, in the sense that i is the argument and the producer to itself) producing *counter-information* c. This fact can be denoted by $c = I(i)$. Counter-information c can now begin to be embedded into information i. Again both, information i and through the Informing I generated counter-information c, have now to generate the information of *embedding* E, by which counter-information c will be embedded into source information i, i.e., $E = i(i, c)$. Embedding information (Informing of embedding) E embeds the arisen counter-information c into already existing information i in such a way that a new total information i_1 arises. This is denoted symbolically by $i_1 = E(c, i)$. In this way the following, so-called minimal informational system is obtained:

$$\begin{aligned} &i;\\ &I = i(i);\\ &c = I(i);\\ &E = i(i, c);\\ &i_1 = E(i, c);\\ &i_1; \end{aligned}$$

This system is called minimal (or the first approximation) because Informing I and embedding E are not considered as argumentative information. As soon as information appears (arises), it can be informationally active (function-like) and/or the argument for other information simultaneously. In this sense, the form $i(i)$ in the second line of the upper system is not trivial. It represents a form of the so-called implicit (implicitly hidden) autoreflexiveness (autoreflexive implicitness) of information. From this line on, the following lines of the system are incomplete in the sense of an adequate I and E consideration. Thus, let us set the following, more completed scheme (the so-called second approximation):

$$\begin{aligned} &i;\\ &I = i(i);\\ &c = I(i, I);\\ &E = i(i, I, c);\\ &i_1 = E(i, c, E);\\ &i_1; \end{aligned}$$

This informational system is enhanced; however, it is far from the so-called maximal approximation. In this, the so-called second approximation there does not appear the explicit autoreflexiveness of information, which will be introduced in the next, the third approximation. Although Informing I and embedding E are autonomous items of information, their functional action is still under supervision (observation) of the entire information i. Consequently, counter-information c arises irrespective of the

produced Informing *I* also through the influence of information *i* and, thus, the third equation of the system becomes *c = I^i(i, I)*. In this equation, ^ represents the symmetrical functional connective (mechanism), where already arisen Informing *I* is under the influence of *i* and information *i* is under the influence of *I*. In general, information is always under the influence of its own arising and vice versa. In this way the later scheme can be advanced into the system (the third approximation)

i;
I = I^I(i, I);
c = I^c(i, I, c);
E = i^E(i, I, c, E);
$i_1 = E\hat{}i_1(i, c, E, i_1);$
$i_1;$

In this scheme the explicit autoreflexiveness of information is coming into the foreground. The autoreflexive explicitness appears in the second, third, fourth, and fifth line of the scheme in the form *I = ...(... I); c = ...(... c); E = ...(... E),* and *i1 = ...(... i1)*. In some way, the implicit autoreflexiveness becomes also more complex, for instance, in *I = i^I(i, I),* information *i* and Informing *I* are implicitly autoreflexive in a more complex (composite, mutually dependent) manner.

For the maximal scheme, the following would hold: everything, which appears informationally as an argument can appear as a function too, and vice versa. The maximal scheme seems to be possible only as an informationally asymptotical concept. Thus, generally, *i(i)* and *i = ...(... i)* are informationally reasonable recursive (implicitly and explicitly autoreflexive) formulae.

It has to be mentioned that in a scheme of the form

i;
i;

the upper and the lower *i* are not equal at all, because they appear separately as *labels* for different informational forms-processes.

6/5

The symbolic notation *i(i)*, which characterizes the arising of information from information by information, is mathematically trivial, inconsistent, because in this case, information *i* is simultaneously an independent and from itself dependent variable. In the rationalistic logic this form of expression is called tautology (the formal agreement of subject and predicate). Within this understanding, information is a tautological principle and tautology itself the fundamental (arising, generative) principle of information. The exclusion of tautology from the classical logical discourse represents a digression from the essential informational research, therefore also from the essential research of logic, mathematics, or any other exact discipline, respectively.

The symbolic notation *i(i)* is tautological, recurrent, and/or recursive. Recursion is a legal mathematical principle where, for instance, the left side of a rule (of substitution or inference) is expressed by its right side. In this case, it would be possible to denote a *rule* by *i := i*. In fact, this rule is recursively trivial, because from a given *i* it generates *i* again, consequently, circulates in itself without arising reproduction. The formula *i := i* illustrates the *waiting*, non-productive cycling, the non-variability of *i*. A more adequate formula would be *i := i(i)*, which already illustrates the previous meaning of the functional tautology *i(i)*.

At this point of discourse, the question can be raised, which information, in fact, arises with the usage of formula *i(i)*. What is the mechanism hidden behind *i(i)*? The principles of arising of information are circularly spontaneous. The mechanism *i(i)* is spontaneous, however, also circular [4], as it proceeds from the previously described schemata of informational arising. The mechanism *i(i)* remains unforeseeable (generally unpredictable), although its unforeseeability is informationally rhythmic (oscillatory) and harmonic (embeddingly adequate, informationally connective, and in this respect, informationally predictable) [5]. That is to say that the rationalistic unforeseeability of the mechanism *i(i)* has just the meaning of informational foreseeing, this is of the arising and embedding of arisen informational products (counter-information) into the existing (valid, original) information. The informational foreseeability (rhythmicalness, harmonicalness) is in no way arbitrarily determinable in advance, but arises also as foreseeability, as the information of foreseeing. The position of the rationalistic tradition is symptomatic exactly in the way it refuses the mechanisms of the type *i(i)*, because by their use each and every rationalistic (informationally closed, hard-defined) discipline would be disintegrated and decomposed.

Common sense and everyday experience approve that the principle *i(i)* is generally valid and that this principle cannot be captured adequately through the standard symbolic, notational, abstract, ideological, logical, mathematical, and even philosophical course of rationalistic thinking. However, this is not true of the philosophy as a whole. Soft philosophies permit circular hermeneutic (a being's semantic) schemata. For these philosophies, statements of the form "language is on the way to language", "information produces information", "a living being is a being's self-production (autopoiesis)", etc. are permissive. *Martin Heidegger's* philosophy is an unsurpassable mastery of tautology. The main reproaches of rationalistic philosophy (materialism, hard dialectic) concern precisely the philosophical tautology. However, in the informational sense, circular inquiring has an essential value, because it does not represent a static (rationalistic) tautology, but can generate intelligence. A soft philosophy enables a spontaneous, free Informing of a being, while a hard philosophy performs as ideology [5], as artificially limiting

informational spontaneity.

7. Differentiation and Counter-Information

7/0

In this section we shall show only one of the several aspects of informational differentiation, namely, the counter-informational one. To understand the mathematical differentiation, let us remember its basic sense and properties.

7/1

Every differentiation has to do with a kind of difference occurring between the source object and that from which this object is derived, i.e., in some way changed one. Differentiation is a procedure by which the derivative product is processed from the original one. This notion coincides with the mathematical concept of differentiation and will, in some manner, coincide also with the informational differentiation. As information, differentiation will be a process of Informing by which the original information will arise (change) into a new form, where the product of differentiation will appear as counter-information. This way of thinking concerning informational differentiation is only a simplified and general one.

Symbolically, if y is a function of x, the instantaneous (non prompt or immediate) rate of change of y with respect to x can be expressed in the form $D_x y$ or dy/dx. This form is called the derivative of y with respect to x. We understand that x in this relation functions as a *reference* information, as a reason (originator) of differentiation.

7/2

Now, let us discuss how counter-information can be brought into the context of differentiation. Counter-information can be understood (within other possibilities) as a product of informational differentiation, i.e., the result which explicates an informationally instantaneous rate of informational arising, changing, and vanishing of information with respect to some specific information. What does informational differentiation as the information which operates along some source-specific information represent in this context? We can choose counter-information as the information-characteristic term for informational differentiation. To continue the course of our discussion, the so-called counter-Informing can be the component distributed (arisingly) in Informing, which produces counter-information. In this sense, we can adopt the following scheme:

source information	informational function to be differentiated
counter-Informing within Informing	process of informational differentiation
counter-information	the derivative: a result of informational differentiation

In the following step, the arisen counter-information has to be (hermeneutically, semantically, meaningly) integrated into the source and into the new information as a whole, otherwise it performs as informational noise. This noise is a process of ignoring or vanishing (dying) of counter-information and can occur already during the cycle of the arising-embedding in which counter-information arose, but the embedding of it was "noisy" by itself. In the next arising-embedding cycle the process of ignoring or vanishing of counter-information continues and so, the effect of informational noise is progressing, causing the losing of substantial counter-informational items.

7/3

On the basis of the previous discussion it is possible to rewrite informational systems (the first approximation) presented in Section 6/4. Let us introduce c for counter-information as

$$c = \mathrm{d}i(i)/\mathrm{d}i$$

which yields

$$\mathrm{d}i(i)/\mathrm{d}i = I(i, C)$$

where C represents counter-Informing within the Informing I. According to this, one can introduce

$$C = \mathrm{d}I(i)/\mathrm{d}i$$

In this manner, according to Section 6/4, the first approximative informational differentiation system becomes

i;
$I = i(i)$;
$c = \mathrm{d}i(i)/\mathrm{d}i$, where
$\mathrm{d}i(i)/\mathrm{d}i = I(i, C)$ and
$C = \mathrm{d}I(i)/\mathrm{d}i$;
$E = i(i, c)$;
$i_1 = E(i, c)$;
i_1;

Similarly, the second approximative system from Section 6/4 can be rewritten into

i;
$I = i(i)$,
$c = \mathrm{d}i(i)/\mathrm{d}i * \mathrm{d}i(i)/\mathrm{d}I$, where
$\mathrm{d}i(i)/\mathrm{d}i = I(i, C)$,
$C = \mathrm{d}I(i)/\mathrm{d}i$,
$\mathrm{d}i(i)/\mathrm{d}I = I(I, C_I)$,
$C_I = \mathrm{d}I(I)/\mathrm{d}I$, and
'$*$' labels a complex informational connective;
$E = i(i, I, c)$;
$i_1 = E(i, c, E)$;
i_1;

and the third approximative system from Section 6/4 has, for instance, the form

i;
I = i^I(i, I);
c = I^(
di(i)/di $*_1$ *di(i)/dI* $*_2$ *di(i)/dc* $*_3$
dI/di $*_4$ *dI/dc* $*_5$
dc/di $*_6$ *dc/dI*
),
where the expression in parentheses *(* and *)* after the sign *'^'* represents *c(i, I, c)* or the taking into account of informationally differentiated (differential) details;
di(i)/dI = I(I, C$_1$), C$_1$ = dI(I)/dI;
di(i)/dc = C$_1$(i, c$_1$), c$_1$ = dC$_1$/dI;
$*_1$, $*_2$, $*_3$, $*_4$, $*_5$, $*_6$ are different forms of complex informational connectives;
E = i^E(i, I, c, E);
i$_1$ = E^i$_1$(i, c, E, i$_1$);
i$_1$;

Similarly, it is not necessary to expose the fact that informational derivatives are always holistically particular (e.g., partial within the mathematical terminology) and that it is, at least informationally, against common sense to construct the so-called total derivation which can exist in mathematics, for in a mathematical function *f(x)*, the variable *x* belongs to a well-defined (definitionally constant) set of objects.

A searching of the so-called maximal general scheme of Informing can be, speaking informationally, only asymptotically approximative. In this context, expressions like *i(i)* or even *di(i)/di* have to be understood informationally holistically, in the sense that in the cases of *i(i)* or *di(i)/di*, in fact, these expressional meanings represent a holistic function or holistic derivative respectively, with *i$_1$(i$_2$)* or *di$_1$(i$_2$)/di$_3$*, where *i$_1$*, *i$_2$*, and *i$_3$* are informational particularities (and thus, mutually divergent, different intentional or chaotic information).

8. Integration and Informational Embedding

8/0

The notion of *integral* represents one of the central notions of mathematical analysis and also of mathematics as a whole. The emerging of this notion concerns two basic tasks: the determination of a function on the basis of its derivative and the calculation of a distance, surface, or volume (generally a multidimensional volume) bounded by the function (its "curve" and coordinates' interval). Two forms arise from this fundamental concept of integral: the *indefinite* and the *definite* ones.

The notion of integral (or integration) in the informational realm is the embedding (incorporation or, mathematically, the operation of finding a function whose derivative is known) of arisen information into the existing information or also a particular form of embedding, which is called reinterpretation (of already embedded information). However, the changing of interpretation can be understood also as informational arising causing a new embedding of an (arisen) informational case. Integration is a very general informational phenomenon taking place during informational arising. *Integration is the most general principle of* informational embedding.

8/1

Every integration has to do with a kind of incorporation of the source object into its broader environment. Integration is a procedure by which the integrated product has been processed from the original, informationally narrower one. Integration has the function of (for instance, hermeneutical or semantical) connection of the arisen (informational) original to a broader (informational) realm. Mathematically, integration produces the function from its derivative and, in this respect, it represents (to some degree) the reversible (or even inversive) operation to mathematical differentiation.

Symbolically, if *y* is a function of *x*, the integral of *y* with respect to *x* can be expressed in the form $INT_{INT\_domain}\ y(x)\ dx$ or in a more compact form as $INT_{d,x}\ y(x)$. In this expression there are two cases of reference: *d* is the *integration domain* (similar to the lower and higher limit) and *x* is the *integration variable.*

8/2

Now, we can discuss the bringing of counter-information into the context of integration. Counter-information as a product of informational differentiation has to be integrated into the already existing, informationally valid information. Otherwise, counter-information is on the way to be vanished, forgotten, and lost as informational noise, which arose, but was not embedded sufficiently into an existing informational realm.

By informational integration, the arisen counter-information is embedded into a closed informational realm and, in this way, the arising of information is, in fact, closed into the cycle of Informing, in which information is coming into existence in an informationally senseful way. The scheme of Informing in 7/2 can now be advanced into the following cycle:

source information	informational function to be differentiated
counter-Informing within Informing	process of informational differentiation
counter-information	the derivative: a result of informational differentiation
counter-information	the reference information for integration into a given informational domain within informa-

nal realm

embedding of counter-information	process of informational integration
new source information	informational function to be differentiated in the next cycle

8/3

On the ground of examples given in 6/4 and 7/3, it is possible to rewrite these systems of Informing considering the function of embedding *E* as a form of informational integration. We have introduced some more detailed, also parallel components of information *i*, Informing *I*, counter-information *c*, counter-Informing *C*, embedding *E*, and information i_1 at the end of arising-embedding cycle into the context of these entities. In this respect, Informing *I* was expressed as

$$I = i(i);$$
$$I = i\hat{}I(i, I);$$

counter-information *c* as

$$c = I(i);$$
$$c = I(i, I);$$
$$c = I\hat{}c(i, I, c);$$
$$c = \mathrm{d}i(i)/\mathrm{d}i;$$
$$c = \mathrm{d}i(i)/\mathrm{d}i * \mathrm{d}i(i)/\mathrm{d}I;$$
$$c = I\hat{}(\mathrm{d}i(i)/\mathrm{d}i *_1 \mathrm{d}i(i)/\mathrm{d}I *_2 \mathrm{d}i(i)/\mathrm{d}c *_3 \mathrm{d}I/\mathrm{d}i *_4 \mathrm{d}I/\mathrm{d}c *_5 \mathrm{d}c/\mathrm{d}i *_6 \mathrm{d}c/\mathrm{d}I);$$

counter-Informing *C* as

$$C = \mathrm{d}I(i)/\mathrm{d}i;$$

and counter-Informing of Informing C_I, which is a component of counter-information *c*, as

$$C_I = \mathrm{d}I(I)/\mathrm{d}I;$$

embedding *E* as

$$E = i(i, c);$$
$$E = i(i, I, c);$$
$$E = i\hat{}E(i, I, c, E);$$

and information i_1 at the end of the arising-embedding cycle as

$$i_1 = E(i, c);$$
$$i_1 = E(i, c, E);$$
$$i_1 = E\hat{}i_1(i, c, E, i_1);$$
$$i_1;$$

Yet, we still have to analyze some possible effects of informational embedding *E*. We can conclude that through embedding some information of embedding *e* and counter-embedding *ec* will also arise. The upper set of expressions for embedding *E* is now coming into consideration in more detail.

8/4

Formally, information of embedding *e* is a sort of *connective* information, by which the arisen counter-information *c* through the process of embedding *E* will be embedded into information *i* obtaining a new informational case i_1. Evidently, information *i*, Informing *I*, counter-information *c*, embedding *E*, information of embedding *e*, and information i_1 are connected information. Information of embedding *e* is the answer to the question how counter-information *c* is connected in i_1. In a realm of information, it is always possible to state the question how information is connected with information or, how one informational case is embedded into another one.

For the information of embedding *e*, the following cases would be possible:

$$e = E(c, i);$$
$$e = INT_i\ c\ \mathrm{d}i;$$
$$e = E(c, i, I);$$
$$e = INT_i\ c\ \mathrm{d}i\ \#\ INT_i\ I\ \mathrm{d}c\ \#\ INT_i\ c\ \mathrm{d}I;$$

where '#' is a complex informational connective.

From these cases we can see that embedding *E* is an operation of integration. In this respect, it is possible to introduce even more complex integrational aggregates, for instance, *E^i*, *E^I*, *E^c*, *E^I^c*, etc. to point out the integrative connectiveness (^) of different informational processes (operations).

8/5

On the basis of the previous discussion, we can rewrite the so-called first approximation of the informational system presented in Section 6/4 and 7/3 in the following way:

$$i;$$
$$I = i(i);$$
$$c = \mathrm{d}i(i)/\mathrm{d}i;$$
$$e = INT_i\ c\ \mathrm{d}i;$$
$$i1 = i\hat{}e(i, c) = INT_i\ e\ \mathrm{d}c\ \#\ INT_e\ c\ \mathrm{d}e;$$
$$i1;$$

In this system we have introduced informational differentiation and informational integration, where counter-information *c* is the derivative of source information *i* (for instance, of a being's metaphysics, state-of-mind of a being), information of embedding *e* is the integral of counter-information *c* in informational domain *i* by *i*, and new information i_1 a complex composite # of integrals, the first of which is the integral of information of embedding *e* in the domain *i* by *i* and the second of which is the integral of information *i* in the domain *e* by *e* (*i-e* mutually dependent embedding).

Similarly, the second approximative system from Section 6/4 and 7/3 becomes

$$i;$$
$$I = i(i);$$
$$c = \mathrm{d}i(i)/\mathrm{d}i * \mathrm{d}i(i)/\mathrm{d}I;$$
$$e = INT_i\ c\ \mathrm{d}i\ \#\ INT_i\ I\ \mathrm{d}c;$$
$$i1 = i\hat{}e(i, c, e) = INT_i\ e\ \mathrm{d}c\ \#1\ INT_e\ c\ \mathrm{d}e\ \#2\ INT_c\ e\ \mathrm{d}i;$$
$$i1;$$

The third approximative system from Section 6/4 and 7/3 has, for instance, the form

```
i;
I = i^I(i, I);
c = I^(di(i)/di *1 di(i)/dI *2 di(i)/dc *3 dI/di *4
                dI/dc *5 dc/di *6 dc/dI;
e = INT1 c di #1 INT1 I dc #2 INTe c dI #3 INT1 i de;
il = i ^1 e ^2 i1(i, I, c, i1) =
                INT1 e dc #1 INTe c de #2 INTe e di #3
                INTe c di1 #4 INT11 c de;
                where ^1 and ^2 are composite
                  informational connectives;
il;
```

8/6

In general, instead of INT_{11} i_2 d_{13}, the denotation (instruction) of the form

integrate i_2 in i_1 by i_3

would be more appropriate, since the *INT* operation has to do with the mathematical integration merely in the cases of the regular mathematical integration. The **integrate** instruction represents informational operation by which the information of embedding (informational connection) is obtained.

9. Category as Information

> *What is* knowledge? *A traditional answer is that knowledge is a form of* justified belief. *... Beliefs can be false, and the truth may not be believed. ... It is of no help to be told that knowledge depends on having* an adequate justification *if, as is so often the case, one is not told what constitutes an adequate justification.*
>
> Fred I. Dretske [1] 85

In the mathematical sense, a category can be understood as the most universal mathematical notion. In such a universe there are certain kinds of objects (these can be represented as informational cases, i.e., informational forms and informational processes) and certain kinds of transformations or morphisms (these can be meant as Informings, i.e., informational processes) between objects. Categories can be conceived as a form of the most general mathematical discourse, the summit of today's mathematical knowledge.

Category K as informational entity can be defined abstractly in the following way:

(1) In a class (a given field of relevance, a generalized set), *informational objects* (information or, more precisely, informational cases in the usual way) will be denoted by capitals *A, B, C, ...* . These objects are understood to be initial information of the categorical class. During the Informing of K, the object class will be generated in the usual informational sense when new objects of the class will arise (counter-information and its embedding), some of them will be changed (counter-informationally and embeddingly), and some of them will disappear (as informationally non-relevant entities). In principle, during the Informing of K, the object class will perform similarly as an informationally generative set.

(2) In a class of Informings, to each pair *A, B* of objects belonging to the object class, a generative set Hom*(A, B)* will be assigned which elements will be called *morphisms*. A morphism is an Informing *I*, which concerns the object pair *A, B*, thus that $B = I(A)$. In this way, object *B* includes the embedded counter-information which arose during Informing *I* of information *A*. The generative set Hom*(A, B)* is a collection of arisen morphisms concerning *A* and *B*.

(3) Abstractly, binary operations of the type

$$\mathrm{Hom}(B, C) \# \mathrm{Hom}(A, B) \dashrightarrow \mathrm{Hom}(A, C)$$

can be performed. According to such operations, the composite morphism, belonging to the set Hom*(A, C)*, denoted by I_2I_1 arises to each pair (I_1, I_2) of morphisms, where I_1 belongs to Hom*(A, B)* and I_2 to Hom*(B, C)*.

For binary operations, the associative law could be introduced: if Informing I_1 belongs to Hom*(A, B)*, I_2 to Hom*(B, C)*, and I_3 to Hom*(C, D)*, then

$$(I_3I_2)I_1 = I_3(I_2I_1)$$

Also, for each object *A*, the so-called waiting (or unit) morphism e_A belonging to Hom*(A, A)* can be introduced, for which

$$I_1e_A = e_AI_1 = I_1$$

holds. However, it is worth mentioning that the associative and the waiting case are idealistic, because, as information, Informings I_1, I_2, and I_3 arise while being composed, and also the waiting before or after Informing may deliver different information.

10. Development of Informational Mathematics: the Mathemation

> *... Tako je jezik odtujevanje od konkretnega carstva reči, oziroma njihove simbolizacije (gledanje na stvar kot na hkratno pojavljanje enega ali več njenih znakov z drugimi), to odtujevanje pa se godi samo za to, da je "snidenje" s konkretnim bolj sijajno ...*[1]
>
> Valter Motaln [8] 176

Mathematics is only one of the possible scientific disciplines concerning abstract symbolic formalization. This way of informational abstraction (rationalism) is called mathematization. In this respect, mathematization is the action of specific

---

[1] ... Thus, language is alienation from the concrete realm of things or from their symbolization (looking to the subject qua to the simultaneous appearing of one or more its symbols among the others), where this alienation takes place only with the aim that the meeting-again with the concrete is shining more ...

abstractness which has the meaning of putting common-sense processes into mathematically legal expressional and semantic (symbolic) forms. In the same way as mathematization concerns mathematics, mathemation (or mathemating) as Informing will concern mathemation (itself).

The new term *mathemation* has, similarly as mathematics, its roots in the Greek *mathema*, which means: *to be learned, learning, knowledge, science.* Mathemation will be a particular, specific field of information representing a formal (or formalistic) *generalization* of the field known as mathematics. Mathemation will include mathematics as a specific rationalistic informational component of mathemation.

If mathematics is explicitly rationalistic, deductive, inductive, mathematical-typically (methodologically) inferential and, in these senses, provable theory, then mathemation is also irrationalistic and has typical properties of Informing, counter-Informing (spontaneous and circular informational generation), and informational embedding (circularly or recurrently spontaneous connectiveness in the realm of meaning). Mathematical notions can always be generalized in some mathemational way, e.g., the notions of logic, relation, set, function, differential, integral, category, etc. can be generalized (notionally broadened) mathemationally.

It is evident that mathemation is in no way a generalization of information. It remains only a part of cultural or a being's informational realm with the purpose of specific, mathemationally characteristic formalization. In the previous sections of this article, it has been shown in which manner certain informational formalizations (in the course of mathematical tradition) are possible. However, this does not mean that completely new ways of formalization, different from mathematical concepts, cannot be sought.

In summarizing the dealt with mathemational notions in the previous sections the following is to be pointed out: informational relation, generative set, functional type *i(i)*, derivative as informational difference, integral as informational embedding, category as informational functional complexity, and last but not least, mathemation as an informationally free principle of abstraction and symbolism are generalized, non-mathematical notions. They exceed mathematically true and legal conventions, however, can be reduced into mathematical objects according to their mathematical conventions as particular, reduced, and simplified (static) cases.

11. Conclusion

Some problems of the rational understanding of information, discussed in this article, represent a set of arguments for the development of a logical system which could be called the *informational logic.* This system has to be constructed in such a manner that arbitrary cases of particular logical systems or logics can be derived on the basis of this generalized concept. In this context, questions of informational truths and falsity, informational derivation and integration, informational modality, inference, particularity, etc. as forms and processes of *informational arising* have to be stated.

In this respect, informational logic deals with the phenomenology of informational arising. In fact, today's logics, from mathematical to modal ones, are studies of ideas concerning linguistic items and dealing exclusively with informationally static, steady, and well-determined informational objects, irrespective of the field of relevance where they are used and of the purposes for which they are constructed. Because of their incapability to express logically arising objects and operations, today's logics are staying outside the possibilities to express dynamic activities appearing in social and man-machine interaction. So far, no logic up to now has tried to capture the most general property of living beings, which appears to be the arising of information as the substantial phenomenology of living self-production and surviving of organisms.

The developing philosophy of information cannot accept uncritically and simplistically the limited and essentially narrowed natures and principles of information as they are proposed under the protection of today's information theory, information science, and even artificial intelligence. It has to search for new logically formalized mechanisms for the expression of structures, whose natures are in the process of coming into existence, changing, vanishing, or disappearing. Informational logic by itself must be conceptualized as a formal system of arising in which new, unforeseeable objects, relations, and operations can come into existence, i.e. can be produced from the existing informational regularity and from the informational chaos.

12. References

[1] *F. I. Dretske:* Knowledge and the Flow of Information. Basil Blackwell Publisher, Oxford (1981).

[2] *A. P. Zeleznikar:* On the Way to Information. Informatica 11 (1987), No. 1, 4-18.

[3] *A. P. Zeleznikar:* Information Determinations I. Informatica 11 (1987), No. 2, 3-17.

[4] *A. P. Zeleznikar:* Principles of Information. Informatica 11 (1987), No. 3, 9-17.

[5] *A. P. Zeleznikar:* Information Determinations II. Informatica 11 (1987), No. 4, 8-25.

[6] *1988 IEEE International Symposium on Information Theory*, Kobe, Japan (June 19-24, 1988) (Call for Papers).

[7] *A. P. Železnikar:* Uvod v formalne informacijske sisteme (An Introduction to Formal Information Systems). Automatika 16 (1975) 3-7.

[8] *V. Motaln:* Jezik in misel (Language and Thought). Anthropos (1985) 166-176.

[9] *G. W. F. Hegel:* The Science of Logic. Hegel's Logic, Translated by W. Wallace with foreword by J. N. Findlay. Oxford, At the Clarendon Press (1985).

[10] *C. V. Negoita:* Expert Systems and Fuzzy Systems. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, Ca (1985).

[11] *S. Kripke:* Semantical Analysis of Modal Logic. Zeitschrift f. math. Logik u. Grundl. d. Mathematik 9 (1963) 67-96.

Problemi racionalnega razumevanja informacije

Pri racionalnem, zlasti matematičnem izražanju informacije kot nastajajočem procesu se pojavljajo značilni problemi matematične nezadostnosti in neustreznosti. Matematični formalizem, ki je bil povsem zadosten za opisovanje dobro definiranih jezikovnih konstrukcij, odpove v tej svoji tradicionalni obliki, ko je potrebno izražati tisto realnost, ki je nastajalno spremenljiva, nepredvidljiva in razvijajoča. Zaradi take (dinamične) informacijske pojavnosti je potreben kritični pogled tudi v same temelje klasične simbolne oziroma formalne logike in opustitev preveč togih in za današnje razmere mišljenja pretrdih, nespremenljivih definicij matematike, kot so npr. sama logika, relacija, množica, funkcija, diferencial, integral, kategorija itd. To seveda ne pomeni, da matematika zgublja svojo dosedanjo vlogo, očitno pa je, da s svojimi nespremenjenimi in nerazširjenimi temelji ne more več konstruktivno in znanstvenorazvojno posegati na nova področja kognitivnega razvoja.

Prispevek v tem članku je seveda še vedno bistveno oddaljen od neke nove, bolj ustrezne formalizacije, kot je klasično matematična. Nekaj stvari pa je vendarle že bolje osvetljenih. Matematična logika, ki izhaja iz mišljenja prejšnjega in iz prve polovice tega stoletja, je zgodovinskorazvojno sicer koristna, vendar je potrebno njene temelje tako preoblikovati, da je iz teh temeljev mogoče brez velikih težav izpeljati različne splošne in konkretnejše logike. Dosedanja formalna logika matematike in njeni različni logični sistemi so v tem konceptu lahko kvečjemu posebni in v svojem bistvu celo bistveno konceptualno reducirani primeri. Iz teksta tega članka je razvidno, kako so posamezni logični, relacijski, množični, funkcijski, diferencialni, integralni, kategorjski primeri pojmov omejeni in neustrezni in tudi, kako bi bila ta njihova dinamična preformulacija mogoča. Ta članek je predvsem mehek uvod v to problematiko in raziskava določenih matematičnih neustreznosti. Cilj te predpriprave pa je seveda izgradnja poti, ki vodi v nekaj, kar bomo v doglednem času lahko imenovali informacijska logika in v okviru katere bo mogoče razviti tudi bistveno drugačne koncepte množice, relacije, funkcije, kategorije itd. kot je v današnji oziroma včerajšnji matematiki to mogoče.

Problematika v tem članku je seveda tudi na poti neke splošne kritike tiste vrste racionalnega razumevanja, ki se največkrat izrojeva v racionalizem in ki skupaj s tradicionalizmom postavlja bistvene blokade napredujočemu mišljenju. Tako tudi za to vrsto racionalnosti velja, da po določenem razvojnem obdobju preide v svojo protiinformacijo, ki postane iracionalnost in za nadaljnji razvoj nesprejemljiva dogmatičnost.

L. M. Patnaik
R. Govindarajan
M. Špegel**
and J. Šilc**
Indian Institute of Science BANGALORE
and Jožef Stefan Institute**

UDK 681.3.001.519.6

The need for parallel processing is felt in many disciplines. In this paper we discuss the essential issues involved in these systems. Various architectural proposals reported in the literature are classified based on the execution order and parallelism exploited. Design and salient features of some architectures which have gained importance are highlighted. Architectual features of non-von Neumann systems such as data-driven, demand-driven, and neural computers, which open the horizont for research in the new models of computations, are presented in this paper. Principles and requirements of programming languages and operating systems for parallel architectures are also reviewed.

Contents

1. Introduction

There has been an ever-increasing need for more and more computing power. In a real-time enviroment, the demands are much more. While the computers built around a single processor cannot stretch their processing speed beyond a few millions of floating point operations per second (FLOPS), an operating speed of a few giga FLOPS is required in many applications. Parallel processing techniques offer a promising scope for these applications.

Several applications such as computer graphics, computer aided design (CAD) of electronic and mechanical systems, wheather modeling and robotics have a high greed for high computing power. To quote an example, ray tracing techniques [32] used to display 3-dimensional objects in computer graphics have to process $5x10^8$ rays, each ray intersecting 10 to 20 surfaces, and each intersection computation requiring 20 to 30 floating point operations on an average. In order to provide a fast, interactive and flicker-free display, each frame has to be computed 30 times per second. Thus, the total computation power required 60 giga FLOPS. Comparing this with the execution speed of the present day supercomputers -- whose expected/measured performance is about 100 millions FLOPS [53] -- we observe that the demand is more by an order of magnitude of two.

The need for parallel processing is conspicuous from the above illustration. The recent advances witnessed in VLSI technology and the consequent decline in the hardware cost further encourage the construction of massively parallel processing systems.

The paper is organized in the following manner. In Section 2, we point out the major issues associated with multiprocessing systems, with a brief discussion on each of these items.

We classify parallel processing architectures broadly into two classes. The first of them, the von Neumann type, includes all multiprocessing systems that adopt the principles of the von Neumann computation model. Pipeline and vector processing, SIMD machines, MIMD machines and VLSI systems which fall under this category are surveyed in Section 3. Supercomputers employ one or more of these techniques for achiving high performance. More emphasis is given to current and recent developments in these areas. In non-von Neumann architecture, we include data-driven, demand-driven and neural computers. Their methodology of computation is functionally different from the ones discussed earlier. An overview of these architectural configurations is presented in Section 4.

The programming language and operating system support required for working with these complex parallel processing systems are reviewed in detail in the subsequent section. Parallel programming languages are classified into two categories: the conventional von Neumann type (that adop the imperative style of programming) and the non-von Neumann languages. Case studies of some of the popular languages and their salient features are also reported.

We remark that an exhaustive coverage of all contributions and research work reported in this fascinanting area of parallel processing systems is beyond the scope of this paper and we do not attempt that. Rather, we will concentrate on the basic principles behind the design of the various architectures. We will pay more attention to some specific architectures and models of computation which have gained importance.

2. Issues in Parallel Processing Systems

In the this section we discuss the various associated with multiprocessing systems. First, it is interesting to look at the models of computation, the way one has evolved from the other, leading to myrial architectural configurations and machines.

2.1. Models of Computation

Depending on the instruction execution order, computer systems can be classified as control-driven, data-driven or demand-driven machines.

Control Flow Conventional von Neumann uniprocessing and multiprocessing systems have an explicit execution order specified by the programmer. This hinders the extent of parallelism that can be exploited. However, these control models perform well for structured data items such as arrays [34]. Also the three decades of programming experience we had with these models still makes it a proponent candidate for future generation computers.

Data-Driven Model In this model, the execution order is specified by the data dependency alone [26]. Instructions are executed as soon as all their input arguments become available to them. Data flow model is suitable for expression evaluation [34]. Since, only data dependency determines the execution order, and the granualarity of parallelism exploited is as low as a single instruction, this model of computation exploits maximum parallelism. However, because of its eagerness in evaluating functions, it executes an instruction, if its operands are available, irrespective of whether or not it is required for the computation of the final result.

Demand-Driven Model In demand-driven (also called reduction) execution, pioneered by Berkling [15] and Backus [9], the demand for result triggers the execution which is turn triggers the evaluation of its arguments and so on. This demand propagation continues until constants are encountered; then a value is returned to the demanding node and execution proceeds in the opposite direction. Since a demand-driven computer performs only those computations required to obtain the results, it will perform less computations, in general, than a data-driven computer. As the computation model is 'lazy'[1] in evaluating expressions, instructions accept partially filled data structures[2]. This makes demand-driven model to support infinite data structures. Such a facility is not available in the other models of computation. However, this model suffers overheads in terms of execution time due to the additional propagation of the demand (for arguments). The control mechanism and the management of program memory add further to the inefficiency [91].

2.2. Concurrency

The granularity and nature of parallelism exploited, significantly influence the performance of the parallel processing systems.

Temporal, Spatial and Asynchronous Parallelism By performing overlapped computation one can exploit temporal parallelism. Pipeline computers [53,77] execute instructions in an overlapped manner as in the assembly line of manufacturing to achieve parallelism. Examples of pipeline processing are the execution of the different phases of an instruction namely, instruction fetch, decode, opcode fetch and execution; execution of the various steps involved in floating point arithmetic operations at various stages of the pipeline. These machines are ideally suited for processing vector instructions [53] namely, vector add, vector multiply and dot product.

Spatial parallelism is the parallelism inherent in performing an operation over the elements of structured data, such as arrays. Spatial parallelism is easy to detect and exploit [34]. The synchronization and scheduling overheads involved in exploiting spatial parallelism are much less compared to those experienced in the other two models.

Multiple instruction multiple data (MIMD) [53] machines achieve asynchronous parallelism through a set of interactive processors with shared resources. Several independent processes running asynchronously on different processors work to accomplish a common goal by exchanging messages or by sharing common variables.

While spatial parallelism is easy to detectet and exploit, the range of applications on which it is inherent is limited. MIMD machines which exploit asynchronous parallelism, on the other hand, offer flexibility in programming. Hence it can be used for a wide range of problems.

Granularity of Parallelism For high performance, we want to exploit as much

---

[1] The nature of the model in executing an instruction only on is termed 'lazy'.

[2] The computation of a recursively defined data structure, for example SEQ(N) = CONS(N,SEQ(N+1)) never terminates, and hence is infinite. Such a data structure can partially be filled by evaluating only those terms which are required for further computation.

parallelism as possible. This means that the granularity of the tasks should be low. We observe that as the level of granularity goes down, the parallelism that can be exploited increases. However, lower the level of parallelism more will be the synchronization overheads. A tradeoff between the parallelism exploited and the synchronization overheads is required to achieve high performance.

Another major issue involved is parallelism detection. Parallelism can either be explicitly specified by the user, or can be detected from a program written in a sequential language using an intelligent compiler. For expressing parallelism explicitly, language such as CSP [50], Occam [55] and Concurrent Pascal [45] can be used. The user need to have a knowledge of the architecture of the machine on which the program is executed, the application program and its runtime characteristics. The second approach which uses program restructuring techniques [71] to transform a sequential program into a parallel form, does not required a knowledged-user. However, these techniques are inefficient and cannot detect parallelism to the fullest extent. Active research to develop efficient parallel programs using these paradigms is underway.

2.3. Scheduling

In a multiprocessor system, scheduling algorithms assign each task to one or more processors with the goal of achieving high performance. Scheduling can again be static or dynamic (refer [6] for a comparison). In static scheduling, the tasks are allocated to processors either during the algorithm design by the user or at compile time by an intelligent compiler. In both these approaches, the scheduling costs are paid only once even if the program is run many times with different data. Morover, there is no runtime overhead. The disadvantage of static scheduling is possible inefficiency in guessing the runtime profile of each task.

Dynamic scheduling at runtime offers better utilization of processors, but at the cost of additional scheduling time. The dynamic scheduling algorithm can be distributed or centralized.

2.4. Synchronization

Synchronization methods are required to coordinate parallel execution of tasks in a multiprocessor system. Architectures with shared storage achive synchronization by semaphores [27] and monitors [49]. In a massage passing multiprocessing system, synchronization of processes is achived using remote procedure calls [46].

In order to update the shared variables in a multiprocessor in a consistent manner (avoiding read-read, read-write races [53]), the updating of the variables is done in a region called critical region. Only one process is allowed to enter the critical region at a time. Preventing access by other processes when the critical region is being accessed by a process is known as mutual exclusion. Synchronization primitives are used to achieve mutual exclusion.

Of the many synchronization primitives proposed (refer [35] for a survey), a few worth mentioning are

(i) the 'test & set' primitive of IBM 360/370 machines [17];
(ii) 'lock' instruction used in C.mmp [101];
(iii) NYU Ultracomputer's 'fetch & add' [42].

The subsequent sections highlight the architectural features of parallel processing systems. The whole spectrum of architectures proposed in the literature can broadly be classified as conventional von Neumann parallel processing systems and the non-von Neumann systems.

3. Von Neumann Parallel Processing Systems

Von Neumann parallel processing systems can be divided into three major architectural configurations: pipeline, SIMD, and MIMD architectures. These three architectural models exploit the three kinds of parallelism, namely temporal, spatial and asynchronous parallelism respectively. Each of the above mentioned architectures is discussed in detail and the design and salient features of certain parallel processing machines are highlighted in this section. Reconfigurable architectures and VLSI systems, though not entirely distinct from the above discussed models, attract the attention of researchers due to their many interesting features. In particular VLSI systems offer new scope in computing for designing dedicated architectures for a variety of applications. Constituent elements of the VLSI systems are systolic [63] and wavefront [64] architectures.

3.1. Pipeline and Vector Processing

Pipeline Architecture Pipelining [77] offers an economical way to realize temporal parallelism. The concept of pipeline processing in a computer is similar to manufacturing in assembly lines in an industrial plant. To achieve pipelining one must subdivide the input task into a sequence of subtasks, each of which can be executed by a specialized hardware stage that operates concurrently with other stages in the pipeline. Successive tasks are pumped into the pipe and get executed in an overlapped fashion at the subtask level. Pipeline processing leads to a tremendous improvement in system throughput. A k-stage linear pipeline could be at most k times faster. However, due to memory conflicts, data dependency, branch and interrupts this ideal speedup cannot be achieved.

One way of classifying a pipeline is based on the function performed by the pipeline. There are two classes of pipelines based on this classification, namely the instruction pipeline and the arithmetic pipeline. In the instruction pipeline, the various phases of instruction execution such as instruction fetch, instruction decode, operand fetch, and instruction execution, are identified and are executed in the successive stages of the linear pipeline. Thus, from pipeline, after an initial delay, instructions are executed once every clock cycle.

Arithmetic pipelines subdivide the arithmetic operations such as floating-point add or floating-point multiply into subtasks and execute them on specialized arithmetic and logic units. In an instruction pipeline, the instruction execution unit can itself be an arithmetic pipeline to further improve the performance. IBM 360/91 machine employs both these pipelines. Much research has already been done on scheduling of pipelines; buffering and delaying techniques are used to improve the execution speed. In Cray 1 [82], there are 12 functional pipeline units to perform both scalar and floating point arithmetic operations. Cyber 205 supports four vector pipelines in addition to a scalar arithmetic

unit.

Vector Processing In this section, we explaint the basic concepts of vector processing [53]. Vector processors operate on vector data and execute vector instructions. Vector pipelines, unlike scalar pipelines, are assured of continuous stream of data. The overheads involved in initializing the vector pipeline is compensated by the speedup improvement gained, as the number of tasks executed is large. Loop termination conditions are performed by specialized hardware in various stage of the vector pipelines. These features make vector processing more efficient than the scalar pipelines.

Vectorizing compilers [6] transform programs written in conventional imperative languages into vector instructions suitable for execution on vector processors. The fact that present day supercomputers have vectorizing compilers and vector processors as their major components demonstrates that pipeline processing is an easy and efficient way of realizing high speed computation. However, not all programs can be vectorized. Pipeline processors perform poorly in such cases.

3.2. SIMD Machines

A synchronous array of parallel processors executing in a lock-step fashion, consisting of multiple processing elements under the supervision of one control unit is called an array processor. The system and user programs are executed under the control of control unit. The array processor can handle single instruction and multiple data (SIMD) stream. SIMD machines exploit spatial parallelism. The processing cells are interconnected by a data-routing network. The interconnection pattern to be established for specific computation is under program control. Vector instructions are broadcast to the processing cells for distributed execution over different processors. The cells are passive devices without instruction decoding capabilities.

Array processors became well publicized with development of Illiac IV [12], Clip 4 [29], and Massively Parallel Processors (MPP) [14]. Future research in SIMD machines is towards designig and implementing multiple-SIMD (MSIMD) machines [53], consisting of more than one control unit. Each control unit in this class of machines shared a pool of dynamically allocatable processors.

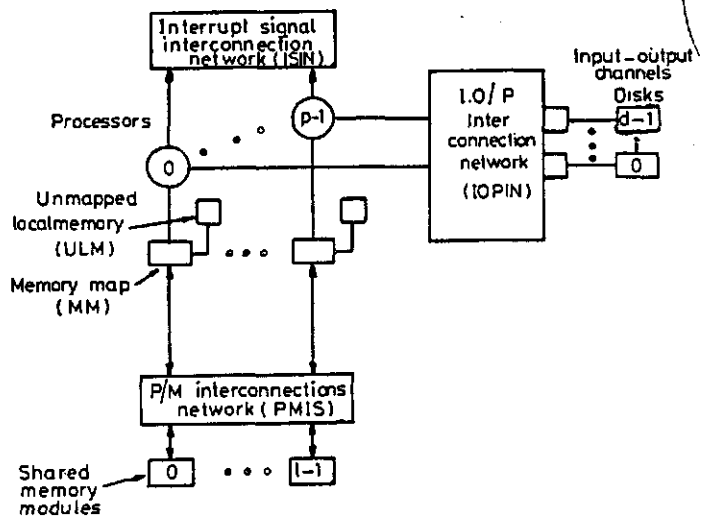
Array processors are characterized by their ability to support parallelism at a low-level. They are ideally suited for specific applications in the areas of image and signal processing [81]. For example, cellular array [81] is a two dimensional array of processors, each of which communicate directly with its neighbors. The structure of the array is very appropriate in terms of layout on a chip. Work in this direction has led us to the design of SIMD architectures for CAD applications [78].

However, as mentioned in section 2.2, the range of applications over which SIMD machines can be put into efficient use is restricted and hence they are not candidate architectures for general purpose parallel processing machines.

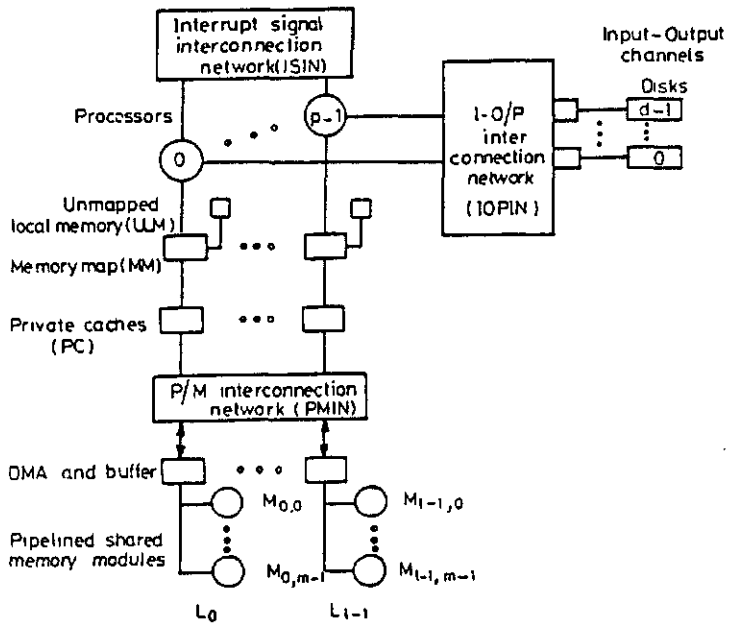
3.3. MIMD Architecture

MIMD machines can be grossly characterized by two attributes: first, a multiprocessor system is a single computer that includes multiple processors and second a multicomputer that has several autonomous computers which are geographically distributed and connected through a communication network. There exists an important distinction between two systems. In the first case, the multiple processors work concurrently in order to achieve a single goal. Interaction between two processors is essentially in terms of intermediate results and synchronization messages. Whereas multicomputers communicate among themselves basically to share expensive resources. Each autonomous computer works on an independent task. In this section we will concentrate more on multiprocessors. Discussion on distributed computing systems using computer network is beyond the scope of this paper.

Multiprocessing systems can be classified into two groups, based on how the processing elements interact among themselves [53]. When several processors communicate among themselves through a shared global memory, we classify them as tightly coupled systems (refer Fig. 1). Hence the rate at which the processors can communicate is of the order of the bandwidth of the memory. On the other hand, we have loosly coupled systems where processors do not share a common memory, but communicate using message passing primitives.



(a) WITHOUT PRIVATE CACHE



(b) WITH PRIVATE CACHE

Fig. 1 Model of a Tightly Coupled System

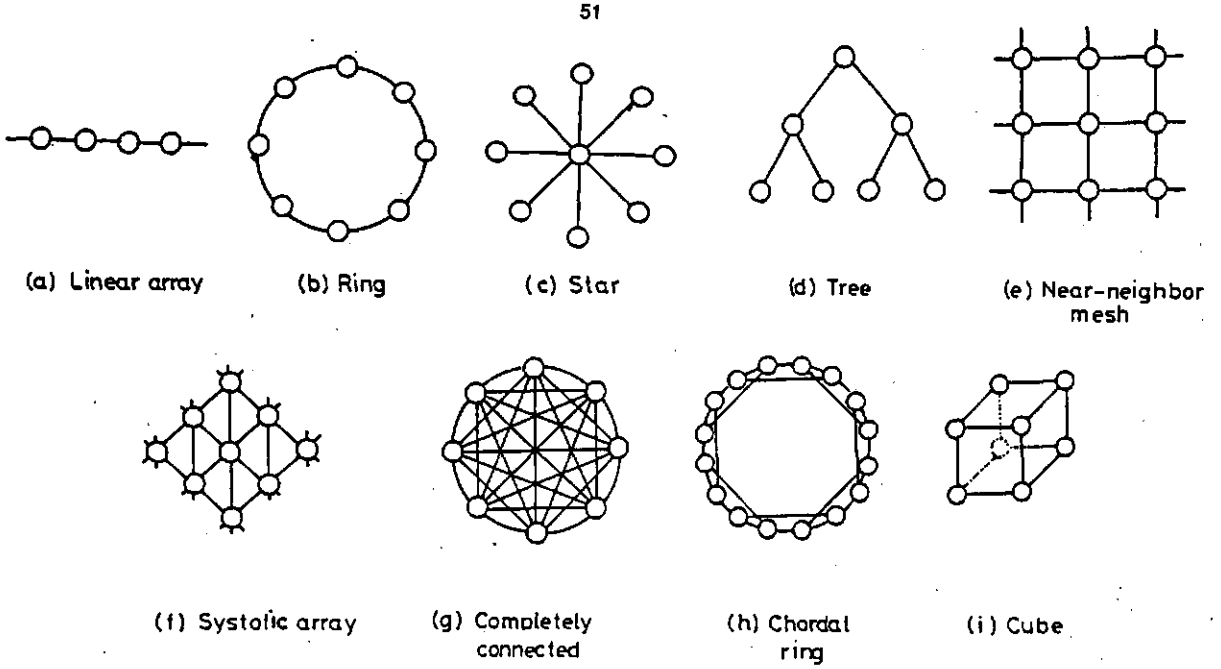


Fig. 2 Static Interconnection Topologies

The interconnection network plays an important role in multiprocessor systems, and significantly influence the performance of the system. A quick overview of the interconnection network is presented in this section, which will be followed by a siscussion on the two configurations of multiprocessing systems.

Interconnection Networks The increasing popularity of many proposed multiprocessing systems with 10^4 to 10^5 processing elements makes the concept, design and implementation of the interconnection network a crucial factor in the design of such systems. A typical interconnection network consists of a set of switching elements. The network can be classified based on the following four design factors: operation mode, control strategy, switching method and network topology [30]. A network can send and receive messages in a synchronous or asynchronous mode. Classification on the control strategy is based on whether the switching elements are controlled in a centralized or distributed manner.

Circuit switching and packet switching are two switching methods adopted in a network. In circuit switching, a physical path is actually established between the source and destination nodes. In contrast, in packet switching, data is put in a packet which is routed through the network without establishing a physical path. Fig. 2 and Fig. 3 depict some of the static and dynamic topologies of computer network.

The bus interconnection scheme connects the various processing cells through a common shared bus. The bandwidth of the bus is very low, and contention is a serious consequence when a large number of cells are connected to the bus. Various schemes [11] such as daisy chain, parallel priority and time-sliced schemes, have been proposed to resolve bus contention. Fig. 4 shows the organization of a shared bus multiprocessing system with daisy chain scheme for priority resolving. Despite its shortcomings, shared bus still attracts system designers because of its low cost and complexity and easy upgradability of the system.

Cross bar switch [53] on the other hand is very expensive, but provides high memory bandwidth. The cost of the network is of the order of n^2, where n is the number of the processing cells in the system.

Multistage interconnection network (MIN) strikes a balance between cost and performance. It is dynamic network using a number of switching elements, unlike a static network where dedicated links are used. A MIN of n x n size establishes a maximum of n links at any instance, at a cost of n log n. This attractive feature makes MIN a proponet in many multiprocessing systems such as the New York university Ultracomputer (NYU) [42].

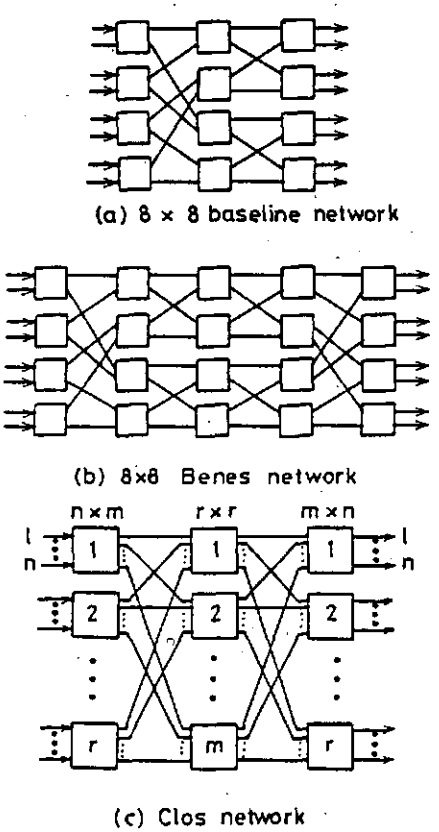


Fig. 3 Dynamic Interconnection Topologies

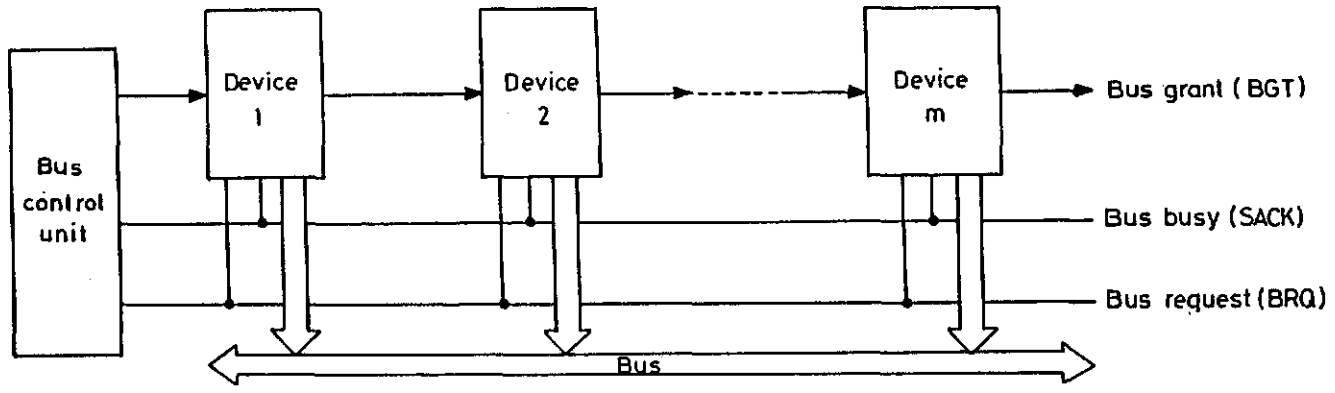


Fig. 4 Daisy Chain Implementation of Shared Bus

Tightly Coupled Multiprocessors Tightly coupled systems are ideally suited if high speed or real-time processing is desired. In a tightly coupled system a set of processing elements is connected to a set of memory modules through an interconnection network.

If two or more processors attempt to access the same memory module, a conflict occurs. Hence the memory modules are either low-order interleaved or high-order interleaved to reduce the number of conflicts. In a variant of tightly coupled systems, each processor is allowed to have a private memory, called the cache for that processor. This will greatly reduce the traffic through the network as local data can be stored in and accessed from the cache.

Processors communicate the intermediate results through the shared memory modules. The set of processors may be homogeneous or heterogeneous. It is homogeneous if the processors are functionally identical. Also, a processor may differ from others in its capability to access I/O systems, performance and reliability.

Various multiprocessing systems have been proposed using a shared memory architecture. Examples of these are, the C.mmp system [101], the Heterogenous Element Processor (HEP) [25], the New York university Ultracomputer (NYU) [42], Honeywell 60/66, Univac 1100/80 and IBM 3084 AP. A dedicated multiprocessing architecture with time shared bus has been designed and experimented by use for computer graphics applications [36,37,38].

Tightly coupled systems can tolerant a higher degree of interaction without much deterioration in performance. However, one of the limiting factor of tightly coupled systems is the performance degradation due to memory conflicts when the number of processors in the system is increased.

Loosely Coupled Multiprocessors Loosely coupled multiprocessor systems do not generally encounter the degree of memory conflicts experienced by tightly coupled systems. In such systems, each processor has a set of input/output devices and large local memory where it accesses most of the instructions and data. Processes which execute on different computer modules communicate by exchanging messages through a message transfer system. The degree of coupling in such a system is very loose (and hence the name). The determining factor in the degree of coupling is the communication topology of the associated message transfer system. Loosely coupled systems are efficient when the interactions among the tasks are minimal.

Fig. 5 illustrates the model of a loosely coupled system. The channel and arbiter switch in each of the computer modules may have a high speed communication memory which is used for buffering block transfers of messages. The communication memory is accessible by all processors through the communication interface. The message transfer system for non-hierarchical system could be a simple time shared bus. The performance of such configuration is limited by the message arrival rate on the bus, the mesage length, and the bus capacity (in bits per second).

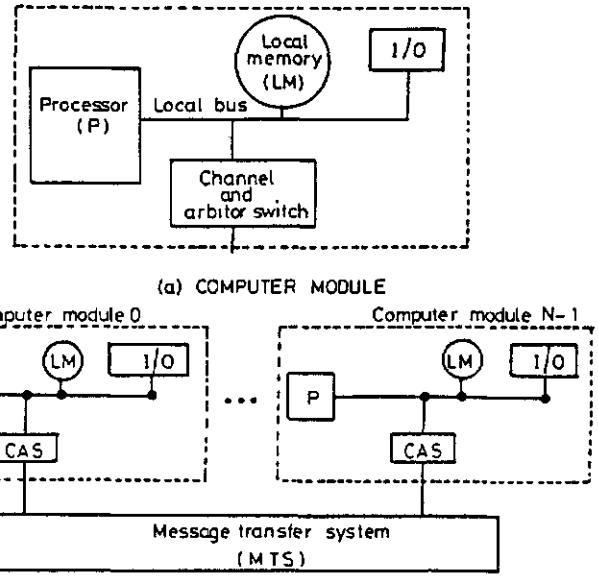


Fig. 5 Model of a Loosely Coupled System

Loosely coupled systems where processors are connected using dedicated links, are also popular. In such a network of processors, locality (nearness between a pair of processors) can be exploited by scheduling tasks which interact among themselves more to a group of processors which are closer to each other. Failure of a node or link will not catastrophically affect the system, as alternate paths can be established and the system can perform with graceful degradation.

The Cm* architecture [54] developed at the Carnegie Mellon University is one example of a loosely coupled system. It consists of a set of clusters connected by an intercluster bus (Fig. 6). Each cluster has a set of processing elements connected over a map bus. The system forms a good example for hierarchical structure.

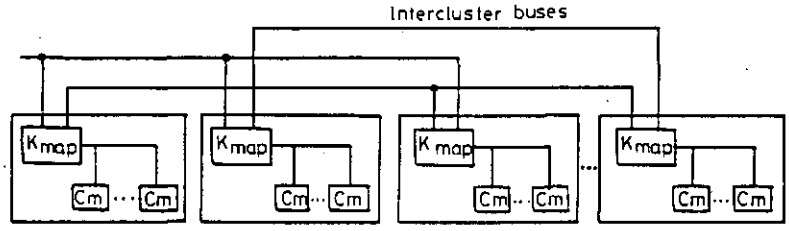


Fig. 6 Cm* Architecture

A loosely coupled system designed for artificial intelligence and image processing application is ZMOB [80]. In this architecture, a set of 256 Zilog microprocessors are connected by a pipeline system.

Another loosely coupled architecture which has attracted the attention of many researchers due to its high performance and relatively low cost, is the hypercube architecture [47,83]. Because of the increased attention it has received, we devote the following paragraphs to this architecture.

The concept of a hypercube computer can be traced back to the work done in the early 1960's by Squire and Palais [93] at yhe University of Michigan. They carried out a detailed paper design of a 4096-node (12-dimensional) hypercube.

The hypercube topology is an n-dimensional generalization of the simple cube. The hypercube of dimension n has 2^n cells. Each cell is connected to n neighboring cells which are at a Hamming distance of one. The connection between adjacent nodes is by point-to-point link. Fig. 7 shows the topology of hypercube architecture for dimensions three and four. The cube manager provides a high level interface for system users. It serves as a local host for the cube, supporting the programming environment, compilation, program loading, input/output and error handling.

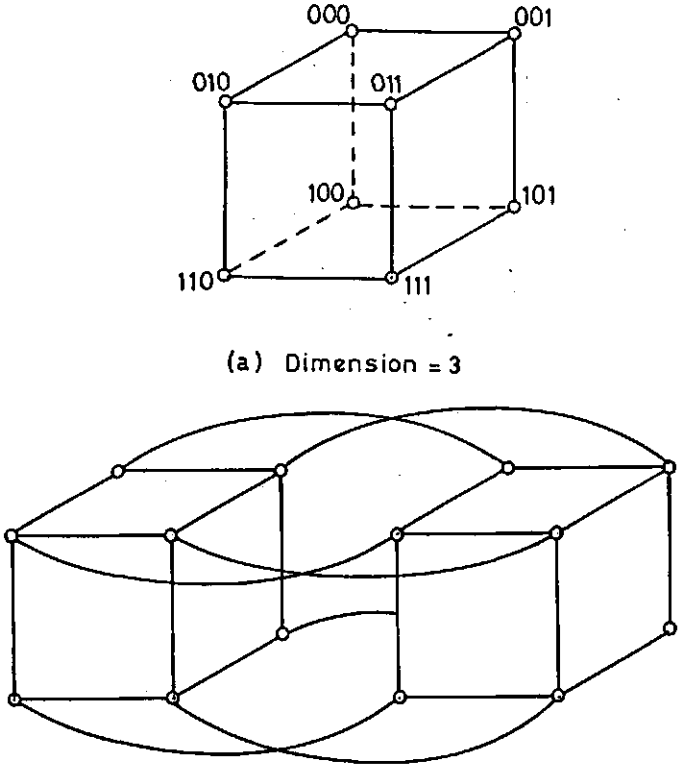


Fig 7. Hypercubes

Hypercube architecture has many attractive properties. The hypercube topology yields a regular array in which the node are close to one another: no more than n steps apart. At the same time, the number of connections from each node to its neighbors is quite low (also equal to n). It thus strikes a balance between a two-dimensional array in which the internode connection costs id low, but the nodes are far apart $O(n^{1/2})$ steps on an average, and a completely connected array in which the internode connection costs is high, but the nodes are only one step apart.

The hypercube architecture is homogeneous in the sense that all the nodes are identical. Further, the hypercube architecture is hierarchical and eminently partitionable. For example, a hypercube of dimension n+1 can be pertitioned into two hypercubes of dimension n. This means that it is quite easy to allocate subcubes to subtasks, especially for problems which adopt a divide-and-conquer strategy. Lastly, the hypercube architecture can itself embed other regular topologies such as tree, mesh, pyramid or hexagonal structure.

3.4. Reconfigurable Architecture

Another interesting aspect of multiprocessing systems in which active research is under progress is reconfigurability [85]. Often, full potentials of multiprocessing systems are not realised in many applications. The reason for this is the mismatch between the application and the architecture. In order to alleviate this problem, we build dedicated systems where the architecture matches the application. Another approach which is actively pursued by researchers is building multiprocessing systems with programmable switches; using these switches it is possible to reconfigure the system depending on the application. Such a reconfigurable system, by nature, is flexible and hence can be used for a variety of applications. Configurable Highly Parallel computer (CHiP) [85] is a reconfigurable system designed to suit the topologies of various applications. In this section we will highlight the salient features of another reconfigurable architecture, the Connection Machine [48]. Its organization is similar to an SIMD machine, but it functions as a reconfigurable architecture executing multiple instructions on multiple data streams, and hence it is hard to classify this as SIMD or MIMD.

Connection Machine The desire to build a machine that will be able to perform the functions of a human mind, the thinking machine, is the motivating force behind the design of Connection Machine. Specifically, retrieving commonsense knowledge from a

semantic network was the application in the designer's mind.

The Connection Machine computes through the interaction of many, say a million, simple identical processing/memory cells. The two requirements for the connection machine are:
(i) each processing element must be as small as possible so that we can afford to have as an many of them as we need;
(ii) the processing elements should be connected by software.

The Connection Machine architecture follows directly from these two requirements. It provides a large number of tiny processor/memory cells connected by a programmable network. Each cell is sufficiently small so that it is incapable of performing meaningful computations on its own. Instead, multiple calls are connected together into data-dependent patterns, called 'active data structures' that both represent and process the data. The activities of these active data structures are directed from outside the Connection Machine by a conventional host computer. This host computer stores data structures on the Connection Machine in much the same way that a conventional machine stores them in a memory. Unlike a conventional memory, though, the Connection Machine has no processor/memory bottleneck. The memory cells themselves do the processing. More precisely, the computation takes place through the coordinated interaction of the cells in the active data structure. Because thousands or even millions of processing cells work on the problem simultaneously, the computation proceeds much more rapidly than would be possible on a conventional machine.

A Connection Machine is connected to a conventional computer much like a conventional memory. Its internal state can be read and written a word at a time from the conventional memory. It differs from a conventional memory in three aspects. First, associated with each cell of storage is a processing cell that can perform local computations based on the information stored in that cell. Second, there exits a general intercommunication network that can connect all the cells in an arbitrary pattern. Third, there is a high-bandwidth input/output channel that can transfer data between the Connection Machine and peripheral devices at a much higher rate than would be possible through the host.

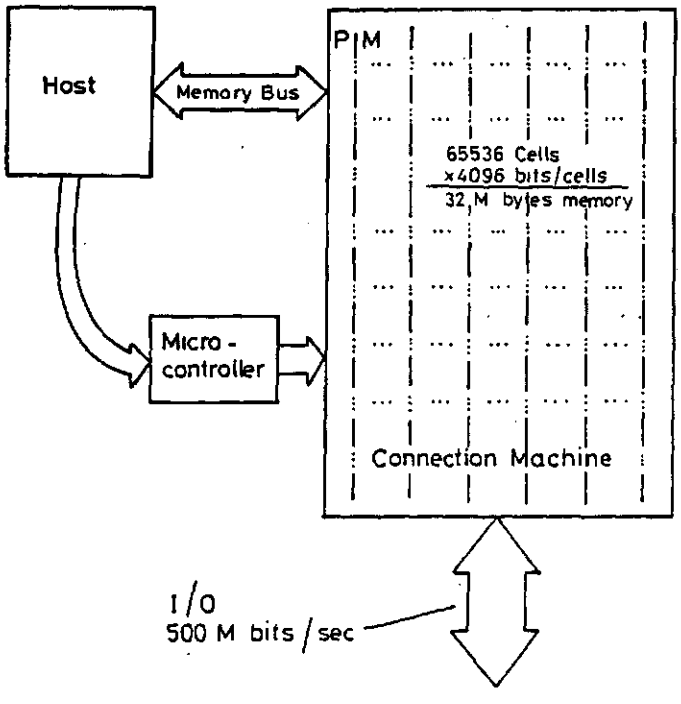


Fig. 8 Connection Machine

A connection is formed between two processing memory cells by storing a pointer in the memory. These connections can be set up by the host, loaded through the input/output channel, or determined dynamically by the Connection Machine itself. In this prototype system, there are 65,536 (2^{16}) processor/memory cells each with 4096 bits of memory. The block diagram of the Connection Machine with host, processor/memory cells, communication network, and input/otput is shown in Fig. 8.

The control of the individual processor/memory cells is orcheastrated by the host of the computer. For example, the host may ask each cell that is in a certain state to add two of its (cell's) memory locations locally and pass the resulting sum to a connected cell through the communication network. Thus a single command from the host can result in tens of thousands of additions and a permutation of data that depends on the pattern of connections. Each processor/memory cell is so small that it is essentially incapable of computing or even storing the results of any significant computation on its own. Instead, the computation takes place in the orchestrated interaction of thousands of cells through the communication network.

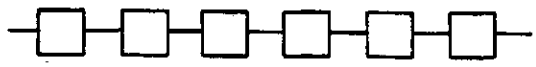
The ability to configure the topology of the machine to match the topology of the problem turns out to be one of the most important features of the Connection Machine. The Connection Machine can also be used as a content addressable or associative memory, but it is also able to perform non-local computations through its communication network.

3.5. VLSI Systems

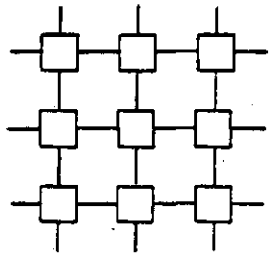
While the previous subsection discusses the features and requirements of reconfigurable architectures, this section is devoted to the architecture of dedicated systems. Nowdays, mature VLSI/WSI (Very Large Scale Integration / Wafer Scale Integration) technology permits the manufacture of circuits whose layouts have minimum feature size of 1 to 3 microns [69]. The effective yields of VLSI/WSI fabrication processes make possible the implementation of circuits with upto half a million transistors at reasonable cost -- even for relatively small production quantities. This opens the horizont for building systems with thousands of processors in a cost-effective and compact manner.

The key attributes of VLSI computing structures [33,63] are
(i) simplicity and regularity,
(ii) concurrency and communication and
(iii) computation-intensiveness.
A VLSI structure should be such that its basic building block is simple and regular, and is used repetitively with simple interfaces. This helps us to cope with high complexity. The algorithm designed for these structures should support a high degree of concurrency, and employ only, simple, regular communication and control to allow efficient implementation. VLSI processors are suitable for implementing compute-bound algorithms. In VLSI processors, we discuss the two classes of architectures namely, systolic and wavefront processors.

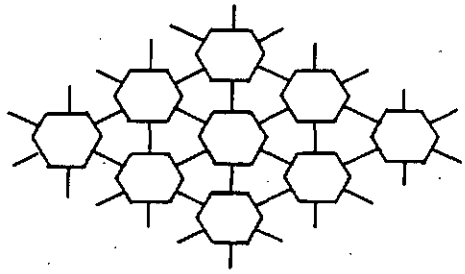
Systolic Arrays Systolic arrays are admired for their elegance and potential for



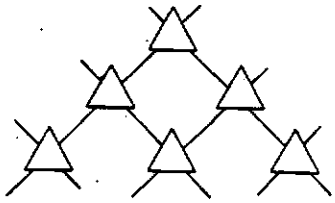
a LINEAR SYSTOLIC ARRAY



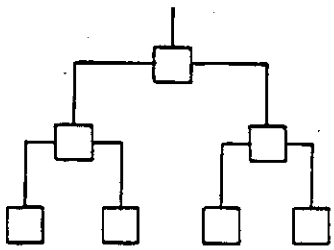
b MESH CONNECTED SYSTOLIC ARRAY



c HEXAGONAL SYSTOLIC ARRAY



d TRIANGULAR SYSTOLIC ARRAY



e TREE STRUCTURED SYSTOLIC ARRAY

Fig. 9 Systolic Arrays

high performance. Systolic arrays belong to the generation of VLSI/WSI architectures for which regularity and modularity are important to achieve area-efficient layouts. The concept of systolic architecture is a general methodology -- rather than being an ad hoc approach -- for mapping high-level computations into hardware structures. In systolic system data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory, much as blood circulates to and from the heart. Systolic arrays derive their computational efficiency from multiprocessing and pipelining. The data items pumped into the systolic processors are reused many times as the move through the pipelines in the array. This results in balancing the processing and input/output bandwidths, a requirement for any parallel processing system for alleviating the von Neumann bottleneck.

The topologies for interconnecting the processing elements of a systolic array are many. Some of the most commonly used topologies, namely linear, mesh, triangular, hexagonal and tree structures are shown in Fig. 9.

Essentially, the whole operation of the systolic system is synchronized with a global clock and it may be visualized as a sequence of computation and data transfer cycles. Incidentally, the clock is the only global signal allowed in systolic architecture apart from the power and ground lines [93]. During the data transfer cycle, all the processing elements pump data into the existing data channels, to be accepted by the neighboring processing elements connected to the data channels. After this cycle is over, all the processing cells enter the computation cycle where each of the cells computes concurrently till the end of the computation cycle. This sequence goes on rhythmicall and perpetually in strict synchronism with the clock beats. Systolic architectures thus capture the concepts of parallel processing, pipelining and regular interconnection structure in a unified framework [63].

Having all the desirable properties of an efficient special purpose system, the systolic architecture is an interesting area of research for a variety of applications, namely digital and image processing [62], linear algebra [28,69], database systems [63], computer graphics [67,96,97], computer-aided design, and solid modeling [60]. The systolic algorithms are characterized by repeated computations of a few types of relatively simpleoperations that are common to many input data items. Often, the algorithms can be described with nested loops or by recurrence equations that describe computations performed on indexed data.

The systolic architectures were originally conceived as devices capable of performing a specialized task. Essentially, these architectures consist of a large number of identical processors, each having only a single arithmetic or logic operation build into its hardware. This greatly limits the applicability of a system to many areas. The latest trend in research in this direction is towards the development of systolic cells which are versatile enough to implement the compute-intensive functions. The design of programmable systolic cells [31] has been suggested as an effective way towards achieving high performance systolic systems. Each systolic cell now possesses a rich instruction set along with some amount of local storage, which were completely absent in the original versions of the systolic architecture. By suitably programming these systolic cells, a variety of operations can be performed. The programmable nature of the systolic cells offers a high degree of flexibility in operation and high performance.

Wavefront Array Processors The data movements in a systolic array are controlled by global timing-reference 'beats'. The burden of synchronizing the operations of the entire systolic computing network becomes heavy for

very large arrays. A simple solution is to take advantage of the data flow computing principle [91] (which will be discussed in detail in Section 4.1), which leads the designer to wavefront array [64] processing.

The wavefront array combines systolic pipelining principle with the data flow computing concept. In fact, the wavefront array can be viewed as a static data flow array that supports the direct hardware implementation of regular data flow graphs. Exploitation of the data flow principle makes theextraction of parallelism and programming for wavefront arrays relatively simpler.

Synchronizing with the global clock and consequently the large surge of current (due to the simultaneous energizing or changing of states of the components) are two major problems in systolic arrays. These can be alliviated in wavefront arrays, because of their asynchronous nature. When the processing times of the individual cells are not uniform, a synchronous array may have to accommodate the slowest cell by using a slower clock. In contrast, wavefront arrays, because of their data-driven nature, do not have to hold back faster cells in order to accommodate the slower one. Wavefront arrays also yield higher speed when the computing times are data-dependent. Lastly, programming of wavefront arrays is easier than that of systolic arrays because wavefront arrays require only the assignment of computations to processing elements, whereas systolic arrays require both assignment and scheduling of computations.

3.6. Critique on von Neumann Systems

The most serious problem with the multiprocessors which use the von Neumann model, as discussed in [10], is the presence of globally updatable shared memory. Special mechanisms are required to ensure correctness of results while updating a memory cell. The explicit execution order to be specified by the programmer is another bottleneck of von Neumann systems. This has led to research in non-von Neumann architectures.

4. Non-von Neumann Architectures

The principal stimuli for developing the non-von Neumann machines have come from the pioneering work on data flow machines by Jack Dennis [26], and on reduction languages and machines by John Backus [9] and Klaus Berkling [15]. In data-driven model, the availability of operands triggers the execution of the operation to be performed on them, whereas in demand-driven model, the requirement for a result triggers the operation that will generate the result. Of late, the realization of the suitability of biological nervous systems for many applications and the use of artificial nets have triggered the design of a new system, the neural computer. In this section, we present the details of these three non-von Neumann approaches.

4.1. Data-Driven Model

In a data flow computer, an instruction is executed as soon as all its operands are available. Since the data availability solely dictates the execution order of instructions, there is no need for having a program counter. Also, the data are passed as values between instructions; this eliminates shared memory and makes the synchronization mechanism simpler. Thus data flow model alleviates the shortcoming of the von Neumann model. Also, parallelism is exploited at instruction level. Hence, a very high speed of computing is possible.

The machine language for a data flow computer is the data flow graph [24]. The data flow graph consists of nodes, to represent operators, and arcs, to carry data between the nodes. Tokens carry data values along the arcs to the nodes. When all the required operands are available, the node is 'fired'. As a result, the input tokens are removed and output tokens are produced.

Data flow architectures are classified as static and dynamic architecture. In static model, an additional constraint -- no token should be present on any of the output arcs of node -- is required to enable the execution of an instruction. The implication of this is that the static data flow model cannot support execution of reentrant routines. This severely restricts the extent of parallelism and asynchrony that can be exploited in the static model. In a dynamic model, additional tags, called environment tags (color), are associated with the token to distinguish the various instantiations of a rentrant routine. Thus, the dynamic model supports fine grain parallelism with full asynchrony. In this paper we describe the architecture of a data flow computer, using the Manchester data flow computer [98] as an illustrative example.

In Fig. 10, the switch unit serves as an interface between the host computer and the data flow machine. It routes the intermediate results to the token queue and the final results to the host computer. The token queue unit is a FIFO buffer which smoothes the flow of tokens in the ring. Tokens in the token queue are checked for their operand type. All tokens directed to single input nodes are directly routed to the node store. Other tokens are sent to the matching unit.

Tokens that arrive in the matching unit search for their matching partner. If the search fails, the token is stored in the matching store; it awaits its partner. On the other hand, if the matching partner is found, it (the matching partner) is removed from the matching store; the incoming token is merged with its partner to form a group token. The group token which contains the information about the operand values, address (of the node to which the token is destined to) and environment tag is sent to the node store.

The node store unit stores the program graph; it stores the opcode for the operator, destination and operand type of result tokens. Tokens entering the node store get the above information to form an executable token. The executable packets are sent to the distributor unit which distributes the tokens to one of the free processing elements. The processing element performs the operation specified by the token to produce result tokens. The result tokens are collected by the arbitrator and are sent to the switch unit. Fig. 10 depicts the block schematic of the Manchester data flow computer.

Research in the area of data flow computation is a rapidly expanding area in United States, Japan and Europe. There are a number of data flow projects that are underway in many universities. Some of them worth mentioning here are M.I.T. data flow computer [26] developed by Dennis, Irvine data flow machine [7,39] by Arvind, Manchester data flow system [98], its extended version (EXtended MANchester architecture) proposed by the

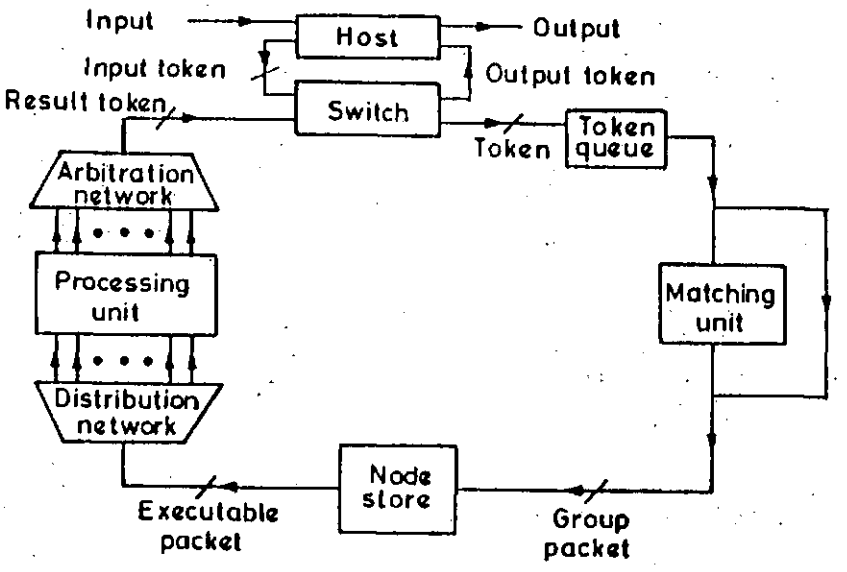


Fig. 10 Block Schematic of Manchester Data Flow Computer

authors [74], Texas Instruments distributed data processor [22], Utah data driven machine [23], Toulouse LAU system [20,76], Newcastle data-control flow computer [89], the efficient static dataflow architecture for specialized computation proposed by the authors [87], and the high speed data flow system developed by Nippon Telegraph and Telephone Systems [3].

Much work needs to be carried out in the design of languages for data flow machines, and implementation of compilers for converting the programs into data flow graphs. (Refer [88] for the initial work on these issues.) Efficient methods to overcome the inherent overheads associated with exploiting fine-grain parallelism have to be developed. Although many data flow machines have been proposed in the literature, no effort is made to prototype them. Demonstrating the feasibility of the data flow model of computation is thus a positive step towards the commercialization of such systems.

4.2. Demand-Driven Systems

In contrast to control flow and data flow programs which are built from fixed-size instructions, demand-driven (reduction) [91] programs are built from nested expressions. The need for result triggers the execution of a particular instruction.

An important point to note is that, in a reduction machine, a program is mathematically equivalent to its value. Demanding the result of definition a, defined as x = (y+1) * (y-z), means that the embedded reference to x is to be rewritten in a simple form. This requires that only one definition of x may occur in a program, and all references to it give the same value, a property known as referential transparency [91].

There are two form of reduction, called string reduction and graph reduction. The basis for string reduction is that each instruction that accesses a particular definition will take and manipulate a seperate copy of the function definition. Whereas, in graph reduction each instruction that accesses a particular definition will manipulate references to that definition. That is, graph reduction is based on sharing of arguments using pointers. In string reduction each access for a definition

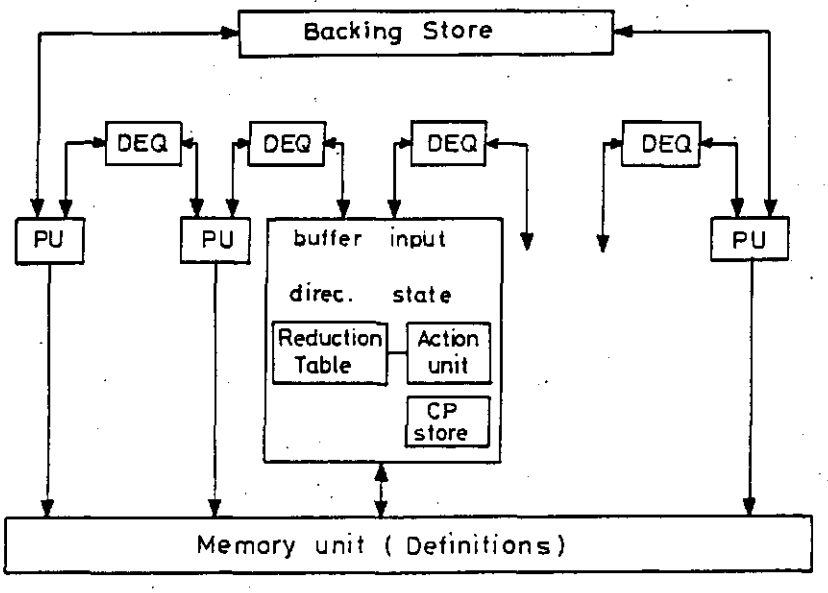


Fig. 11 The Newcastle Reduction Machine

will result in the evaluation of the definition. Reduced definitions or data values are accessed when the demanded definition has already been evaluated. While graph reduction machine takes advantage of the shared definition (in term of the number of definitions evaluated), it is more complex than string reduction.

There are two basic problems in supporting reduction approach on a machine organization [90]: first, managing dynamically the memory of the program structure being transformed and, second, keeping control information about information about the state of the transformation. The basic organization of a reduction machine, the Newcastle reduction machine [90], is presented below (refer Fig. 11)

The reduction machine organisation discussed here supports reduction by expression evaluation. To find work, each processing element traverses the subexpression in its memory looking for a reducible expression. When a processing element locates a reference to be replaced by the corresponding definition, it sends a request to the communication unit via its communication element. The communication unit in such a computer frequently organized as a tree-structured network on the assumption that the majority of communications will exhibit properties of locality of reference. Concurrency in such reduction computers is related to the number of reducible subexpressions at any instant and also to the number of processing elements that traverse these expressions.

Apart from the pioneering work of Klaus Berkling [15], the stage of development of reduction computers somewhat lags behind that of data flow computers [91]. However, researchers have demonstrated the principle and feasibility of reduction machine organization by designing many prototype system such as the GMD reduction machine [58], Newcastle reduction machine [90], Mago's cellular tree machine [66], Applicative Multiprocessing Systems (popular known as AMPS) [57], and Combridge University SKIM reduction machine [92].

4.3. Neural Computers

In the fields of image processing and speech recognition, the ability to adapt and continue learning is essential. Traditional techniques used in these applications are not adaptive. It has been realized that the biological nervous system is more suitable for applications involving pattern recognition and learning [2]. Artificial neural nets have been studied during the last few years in the hope of achieving human like performance in the fields of image processing and speech recognition. The neural net models [65] attempt to achieve good performance via dense interconnection of simple computational elements. The interest in this type of non-von Neumann computing techniques in recent years is due to the development of new net topologies and algorithms, new analog VLSI implementation techniques and the growing fascinantion for the understanding of the functioning of the human brain as well as the realization that human-like performance is required for applications involving enormous amount of processing [51,65]. Several mathematical models have been proposed to exhibit some of the essential qualities of human mind: the ability to recognize patterns and relationships, to store and use knowledge, to reason and plan, to learn from experience and to undrstand what is observed.

Neural net models are specified by the net topology, node characteristics and training or learning rules. The computational elements or nodes used in neural net models have nonlinear characteristics, typically analog, and are specified by the type of nonlinearity. Most net algorithms also adapt in time to improve performance based on current results. Any artificial neural model must necessarily be a speculation: definitive experimental evidence about the structure and function of the neurological circuitry in the brain is extremely difficult to obtain since it is hard to measure the neural activity without interfering with the flow of information in the neural circuit. Further, the neurons are intricately interconnected and the flow of information is complicated by the presence of multiple feedback loops.

Nevertheless enough is known about some parts of the brain to fuel the desire for constructing mathematical models of the neural circuit. In general, the models propose to generate a sensory-activated goal-directed behavior and control a multilevel hierarchy of computing modules. At each level of hierarchy, input commands are decomposed into strings of output subcommands, that form input commands to the next lower level. Feedback from external environment or from internal sources drives the decomposition process, and steers selection of subcommands to achieve the goal successfully (refer Fig. 12).

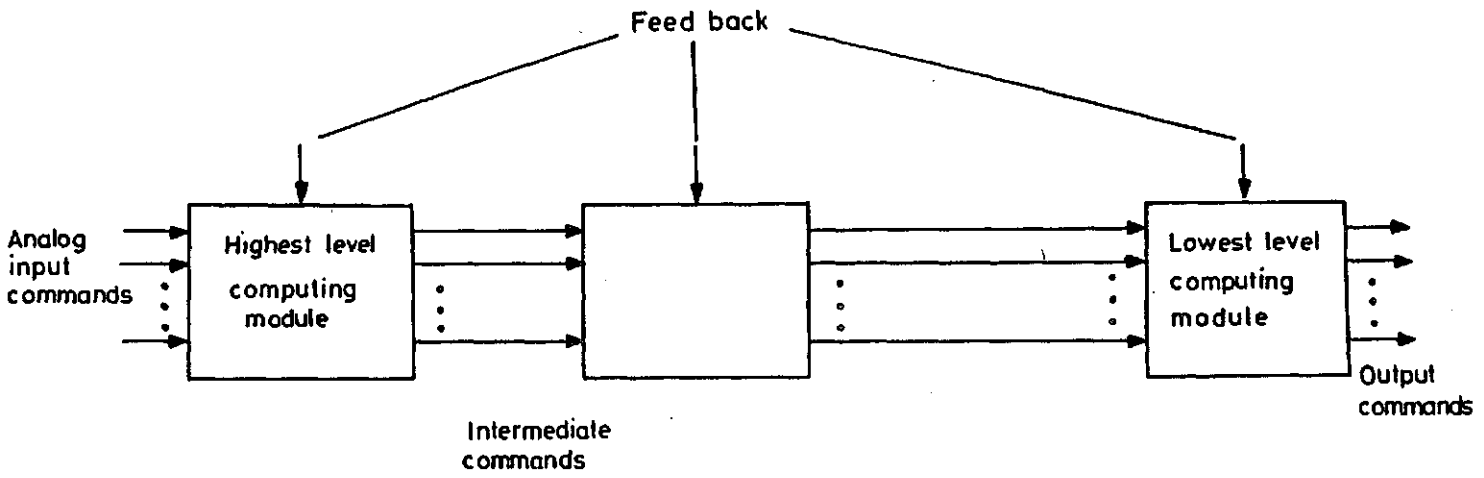


Fig. 12 Neural Net Modules

The benefits of neural net include high computation rates, provided by massive parallelism and a greater degree of fault-tolerant since there are many processing nodes each with primarily local connections. Designing artificial neural nets to solve problems and studying real biological nets may also change the way we think about problems and lead us to new insights and algorithmic improvements.

5. Software Issues Related to Multiprocessing Systems

Having discussed the various multiprocessing systems, it is worth probing further into two aspects of parallel processing: the language to program and the operating system support to handle these complex systems.

5.1. Parallel Programming Languages

One of the motivations behind the development of concurrent languages has been the structuring of software -- in particular, operating system -- by means of high-level language constructs. The need for liberating the production of real-time applications from assembly language has been another driving force. In the following discussion, we classify the concurrent high-level languages into two groups: 'traditional' languages of the von Neumann type (based on imperative style of programming) and the unconventional languages, such as data flow, functional and logic-based languages. A quick review of some of these languages is presented below.

Conventional Parallel Languages Based on the imperative style of programming, these languages are just the extensions of their sequential counterparts. A concurrent language should allow programmers to define a set of sequential activities to be executed in parallel, to initiate their evolution and to specify their interaction (refer [4] for an excellent survey on the concepts and notations for parallel programming). One important point regards the 'granularity of parallelism', i.e. the kinds of granules that can be processed in parallel. Some languages specify concurrency at statement level, and certain others at task level. Constructs for specifying inter-activity interaction are probably the most critical linguistic aspects of concurrency. Language constructs ensuring mutual exclusion are called synchronization primitives. Some of the best-known and landmark solutions that have been adopted to solve these problems are the semaphores [27], mailboxes, monitors [49] and remote procedure calls [46]. Research in this direction is towards designing new machanisms for interprocessor communication, such as ordered ports [13]. In the following discussion we restrict ourselves to a few parallel programming languages and their salient features.

Communicating Sequential Processes (CSP) [50] is a language designed especially for distributed architectures. In CSP, activities communicate via input/output commands. Communication requires both the participating processes to issue their commands. Also CSP achieves process synchronization using the input/output commands. Another interesting feature of this language is its ability to express non-determinism using guarded commands. An implementation of a subset of CSP [73] has been has been successfully attempted by the authors' research group. Their work on the design of an architecture to execute CSP is reported in [79].

Distributed Processes (DP) [46], developed by Hansen, is proposed for real-time applications controlled by microcomputer networks with distributed storage. In DP, a task consists of a fixed number of subtasks that are started simultaneously and exist forever. A process can call common procedures defined within other processes. These procedures are executed when the other processes are waiting for some conditions to become true. This is the only means of communication among the processes. Processes are synchronized by means of non-deterministic guarded commands.

Occam language [55], which is based on CSP has also been designed to support concurrent applications by using concurrent processes running in a distributed architecture. These processes communicate through channels. The Transputer [56], also developed by Inmos Corporation, supports the direct execution of this language.

Languages which adopt monitor-based solution for synchronization are oriented towards architectures with shared memory. Examples of these languages are Concurrent Pascal [45], Ada [5], and Modula [99]. These languages, in general, support strong type checking and seperate compilation, and express concurrent actions using explicit constructs.

Concurrent Pascal [45] extends the sequential programming language Pascal using the concurrent programming tools, processes and monitors. The main contribution of this language is extending the concept of the monitor using an explicit hierarchy of access rights to shared data structures that can be started in the program text and checked by a compiler.

The programming language Modula [99] is primarily intended for programming dedicated computer systems. This language borrows many ideas from Pascal, but in addition to conventional block structure, it introduces a so-called module structure. A module is a set of procedures, data types and variables where the programmer has precise control over the names that are imported from and exported to the environment [99]. Modula includes general multiprocessing facilities such as processes, interface modules and signals.

In Ada [5], we only have active components, the tasks. Information may be exchanged among tasks via entries. An entry is very similar to a procedure. The call to an entry is like a procedure call; parameters should be passed if the called entry requires them. A randezvous is said to occur when the caller is in the call state and the called task is in the accept state. After executing the entry subprogram both the tasks resume their parallel execution. Ada provides specific and elaborate protocols for task termination. Ada is designed to support reliable and efficient real-time programming.

Non-von Neumann Languages Conventional languages imitate the von Neumann computer. The dependency of these languages on the basic organization of the von Neumann machine is essentially a limitation to express and exploit parallelism [10]. These imperative languages perform a task by changing the state of a system rather than modifying the data directly. In parallel processing applications, it makes more sense to use a language with a

nonsequential semantic base. Various paradigms have been adopted and new programming languages based on these approaches have evolved. We will restrict ourselves in this paper to two such paradigms, namely the applicative and the non-procedural style of programming, and the resulting parallel versions of the languages that adopt these approaches.

Applicative languages (also referred to as functional languages) avoid side-effects, such as those caused by an assignment statement. The lack of side-effects accounts, at least partially, for the well-known Church-Rosser property, which essentially states that no matter what order of computation is chosen in executing a program, the program is guaranteed to give the same result (assuming termination). This marvellous determinacy property is invaluable in parallel systems. Another key point is that in functional languages the parallelism is implicit and supported by their underlying semantics.

A system of languages known as Functinal Programming languages (FP) [10] and Lisp [68] are two major outcomes of the applicative style of programming. Languages for data flow architectures, which avoid side-effects and encourage single assignment, are also included in the set of applicative languages. Dennis' Value oriented Algorithmic Language (VAL) [1], Arvind's Irvine Data flow language (Id) [8], and Keller's Flow Graph Language (FGL) [57] are candidate examples in this category.

Considerable work has been done by us in the area of aplicative programming languages. A high level language for data flow computers, called Data Flow Language (DFL) [72], has been proposed by us, and a compiler to convert the programs written in this language into data flow graphs has been implemented. The concepts borrowed from CSP and DP when embedded into data flow systems results in two new languages for distributed processing, namely Communicating Data Flow Channels (CDFC) and Distributed Data Flow (DDF) respectively [75]. Communication and non-determinism features have been added to FP by us [40,41] to strengthen its power as a programming language. We have also proposed that FP can be used as a language for program specification [41].

Although parallelism in a program is expressed by the functional languages in a natural way, their automatic detection and mapping to processors do not result in optimal performance. It is desirable to provide the user with the ability to explicitly express parallelism and mapping, retaining the functional style of programming. Languages which allow the programmers to annotate the parallelism and mapping scheme for the target architecture lead to optimal performance on a particular machine. Two languages developed with this motivation are ParAlfl (the Para-Functional language) [52] and Multilisp [44]. Efforts have been taken to exploit the advantages offered by the functional languages to the maximum extent by developing new machines based on non-von Neumann architecture (refer [95] for a recent survey).

Applications such as the design of knowledge base systems and natural language processing revealed the inadequacies of the conventional programming languages to offer elegant solutions. The use of predicate logic, which is a high-level human oriented language for describing problem and problem solving methods for computers, promised great scope for these applications. Logic programming languages combine simplicity with a lot of powerful features. They separate the logic and control [59], the two major components of an algorithm, and thus enable the programmer to write more correct, more easily improved and more readily adapted programs. The powerful features of these languages also include the declarative style, the unification mechanism for parameter passing and execution strategy offered by non-deterministic computation rule. The powerful execution mechanism provided by these languages is due to the non-procedural paradigm. An outcome of the research carried out in this area with these motivations is the design of the language Prolog [19].

Logic programming languages offer three kinds of parallelism, namely the 'AND', 'OR' and 'argument' parallelism [21,43]. The inability of the von Neumann architecture to efficiently execute logic programming language (in essence supporting non-procedural paradigm) has led to the design of many parallel logic programming machines [16,43,70,94,100]. Further research on these languages has led to the design of three parallel logic programming languages, the PARLOG (PARallel LOGic programming) [18], P-Prolog [103] and Concurrent Prolog [84].

With the increasing size and complexity of parallel processing systems, it becomes essental to design efficient operating systems, without which handling of such systems would be impossible. The general principles and requirements of multiprocessor operating systems are discussed in the following section.

5.2. Multiprocessor Operating Systems

The basic goals for an operating system are to provide programmer-interface to the machine, manage resources, provide mechanisms to implement policies, and facilitate matching applications to the machine. There is conceptually little difference between the operating system requirements of a multiprocessor and those of a large computer system with multiprogramming. The operating system for a multiprocessor should be able to support multiple asynchronous tasks which execute concurrently, and hence is more complex.

The functional capabilities of a multiprocessor operating system include resource allocation and management schemes, memory and data set protection, prevention of system deadlock and abnormal process termination or exception handling and processor load-balancing. Also, the operating system should be capable of providing system reconfiguration schemes to support graceful degradation of performance in the event of a failure. In the following discussion, we introduce brifly the three basic configurations, namely master-slave, separate supervision and floating-supervision systems [53].

In a 'master-slave' configuration, one processor, called the master, maintains the status of all processors in the system and apportions the work to all the slave processors. Service calls from the slave processors are sent to the master for executive service. Only one processor (the master) uses the supervisor and its associated procedures. The merit of this configuration is its simplicity. However, parallel processing system which has the master-slave configuration is susceptible to catastrophic failures, and a low utilization of the slave processors may result if the master cannot despatch processes fast enough. Cyber 170 and DEC System 10 use this mode of operation. The master-slave

configuration is most effective for special applications where the work load is well-defined.

In a 'seperate supervisor system', each processor contains a copy of a basic kernel. Each processor services its own needs. However, since there is some interaction among the processors, it is necessary for some of the supervisory code to be reentrant, unlike in the master-slave mode. Separate supervisor mode is more reliable then master-slave mode. But the replication of the kernel in all the processors causes an under-utization of memory.

The 'floating supervisor' scheme treats all the processors as well as other resources symmetrically or as an anonymous pool of resources. In this mode, the supervisor routine floats from one processor to another, although several of the processors may be executing service routines simultaneously. Conflicts in service requests are resolved by priorities. Table access should be carefully controlled to maintain the system integrity. The floating supervisor mode of operation has the advantages of providing graceful degradation and better availability of reduced capacity systems. Furthermore, it is flexible and it provides true redundancy and makes the most efficient use of available resources. Examples of the operating systems that execute in this mode are the MVS and VM in the IBM 3081 and the Hydra [102] on the C.mmp.

6. Conclusions

In this survey, we have identified the various issues involved in parallel processing systems. Approaches followed to solved the associated problems have also been discussed and their relative merit are put forth. The principles and the requirements of language and operating system support for complex multiprocessing systems are elaborately described. For the wide spectrum of architectures proposed in the litarature, their design principles and salient features are brought out in a comparative menner.

While the envisaged potentials offer a promising scope for parallel processing systems for many applications, hardly a few systems are commercialized. The reasons fot this is the lack of good software support for these systems. Design of intelligent compilers which can identify parallel subtasks in a program (written in a sequential language), schedule the subtasks to the processing elements and manage communication among the scheduled tasks, is a step toward this end. Although there are many existing proposals in this line, none of them seems to achive all the three goals in an integrated manner, relieving the burden from the user completely.

Another question that remains unanswered is whether or not to continue with von Neumann approach for building complex parallel processing machines. While familiarity and the past experience with control flow model make it a proponent candidate, its inherent inefficiencies, such as the explicit specification of control and global updatable memory, limit its capabilities. Although data-driven and demand-driven computers exploit maximum parallelism in a program, their complex structure and inadequate software support force the designer to have a second throught on these approaches.

With the advent of VLSI technology and RISC design, dedicated architectures are becoming more and more popular. However, the inapplicability of these systems to a variety of applications causes a serious concern. At the other end of the spectrum, we have general purpose parallel processing systems which give degrade performance due to the mismatch of the architecture and algorithm, and the reconfigurable machines. Considerable and on design efficient algorithms (for general purpose computing systems) which will bridge the gap between the application program and architecture.

Finally, the research on neural computer and molecular machines is at its infancy. Modeling the neural circuits and understanding the functioning of human brain have to be considerable refined before one could make use them for building high speed computing systems.

The vastness of this fascinanting area in which active research is underway, and the innumerable problems that remain to be solved are themselves standing evidences for the promising future of parallel processing. With the ever-growing greed for very high speed of computing, and with the inability of the switching devices to cope up with the need, parallel processing techniques seem to be the only alternative.

7. References

1. W.B. Ackerman and J.B. Dennis, "VAL - A Value Oriented Algorithmic Language, Preliminary Reference Manual", Tech. Rep. TR-218, Lab. for Computer Science, M.I.T. Cambridge, MA, June 1979.

2. J.S. Albus, Brains, Behavior and Robotics, Byte Books, McGraw-Hill, New York, NY, 1981.

3. M. Ammamiya, R. Hasegawa and H. Mikami, "List Processing and Data Flow Machines", Lecture Note Series, No.436, Research Institute for Mathematical Sciences, Kyoto University, September 1980.

4. G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming", Computing Surveys, Vol.15, No.1, pp.3-43, March 1983.

5. ANSI/MIL-STD 1815A, Reference Manual for Ada Programming Language, 1983.

6. C.N. Arnold, "Performance Evaluation of Three Automatic Vectorizer Packages", Proc. Int'l Conf. Parallel Processing, pp.235-243, 1982.

7. Arvind and K.P. Gostelow, "A Computer Capable of Exchanging Processors for Time", Proc. IFIP Congress, pp.849-854, 1977.

8. Arvind, K.P. Gostelow and W. Plouffe, "An Asynchronous Programming Language and Computing Machine", Tech. Rep. TR-114a, Dept. Information and Computer Science, University of California, Irvine, CA, December 1978.

9. J. Backus, "Reduction Languages and Variable Free Programming", Rep. RJ.1010, IBM T.J. Watson Research Center, Yorktown Heights, NY, April 1972.

10. J. Backus, "Can Programming be Liberated from von Neumann Style? A Functional Style and its Algebra of Programs", Communications of the ASM, Vol.21, No.8, pp.613-641, August 1978.

11. W.L. Bain Jr. and S.R. Ahuja, "Performance Analysis of High-Speed Digital Busses for Multiprocessing Systems", Proc. 8th Ann. Symp. Computer Architecture, pp.107-131, May 1981.

12. G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick and R.A. Stokes, "The ILLIAC IV Computer", IEEE Trans. on Computers, Vol.C-17, No.8, pp.746-757, August 1968.

13. J. Basu, L.M. Patnaik and A.K. Goswami, "Ordered Ports - A Language Construct for High Level Distributed Programming", The Computer Journal, Vol.30, 1987.

14. K.E. Batcher, "Design of a Massively Parallel Processor", IEEE Trans. on Computers, Vol.C-29, No.9, pp.836-840, September 1980.

15. K. Berkling, "Reduction Languages for Reduction Machines", Proc. 2nd Int'l Symp. Computer Architecture, Januar 1975.

16. L. Bic, "A Data-Driven Model for Parallel Implementation of Logic Programs", Dept. Information and Computer Science, University of California, Irvine, CA, January 1984.

17. R.P. Case and A. Padegs, "Architecture of the IBM System 370", Communication of the ACM, Vol.21, No.1, pp.73-96, January 1978.

18. K. Clark and S. Gregory, "PARLOG: PARallel Programming in LOGic", ACM Trans. on Programming Languages and Systems, Vol.8, No.1, pp.1-49, January 1986.

19. W.F. Clocksim and C.S. Mellish, Programming in Prolog, Springer-Verlag, New York, NY, 1984.

20. D. Comte, A. Durrieu, O. Gelly, A. Plas and J.C. Syre, "TEAU 9/7 SYSTEME LAU - Summary in English", CERT Tech. Rep. #1/3059, Centre d'Etudes et de Recherches de Toulouse, October 1976.

21. J.S. Conery and D.F. Kibler, "AND Parallelism and Nondeterminism in Logic Programs", New Generation Computing, Vol.3, No.1, pp.43-70, 1985.

22. M. Cornish, "The TI Data Flow Architectures: The Power of Concurrency for Avionics", Proc. 3rd Conf. Digital Avionics Systems, pp.19-25, November 1979.

23. A.L. Davis, "The Architecture and System Method of DDM1: A Recursive Structured Data Driven Machine", Proc. 5th Ann. Symp. Computer Architecture, pp.210-215, April 1978.

24. A.L. Davis and R.M. Keller, "Data Flow Graphs", Computer, Vol.15, No.2, pp.26-41, February 1982.

25. Deneclor, Inc. Heterogeneous Element Processor: Principles of Operation, April 1981.

26. J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor", Proc. 2nd Int'l Symp. Computer Architecture, pp.126-132, January 1975.

27. E.W. Dijkstra, A Discipline of Programming, Prentice-Hall Inc., Englewood Cliffs, NJ, 1976.

28. B.L. Drake, F.L. Luk, J.M. Speiser and J.J. Symguski, "SLAPP: A Systolic Linear Algebra Parallel Processor", Computer, Vol.20, No.7, pp.45-49, July 1987.

29. M.J. Duff, "CLIP-4", in Special Computer Architecture for Pattern Recognition, K.S. Fu and T. Ichigawa (Eds.) CRC Press, 1982.

30. T.Y. Feng, "A Survey of Interconnection Networks", Computer, Vol.14, No.12, pp.12-27, December 1981.

31. A.L. Fisher, H.T. Kung, L.M. Monier, H. Walker and Y. Dohi, "Architecture of the PSC: A Programmable Systolic Chip", Proc. 10th Ann. Symp. Computer Architecture, 1983.

32. J.D. Foley and A. van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley, Reading, MA, 1982.

33. J.A.B. Fortes and B.W. Wah, "Systolic Arrays -- From Concept to Implementation", Guest Editor's Introduction, Computer, Vol.20, No.7, pp.12-17, July 1987.

34. D.D. Gajski and J. Peir, "Essential Issues in Multiprocessor Systems", Computer, Vol.18, No.6, pp.9-27, June 1985.

35. C. Ghezzi, "Concurrency in Programming Languages: A Survey", Parallel Computing, Vol.2, pp.229-241, 1985.

36. D. Ghosal and L.M. Patnaik, "Performance Analysis of Hidden Surface Removal Algorithms on MIMD Computers", Proc. IEEE Int'l Conf. on Computers, Systems and Signal Processing, pp.1666-1675, December 1984.

37. D. Ghosal and L.M. Patnaik, "Parallel Polygon Scan Conversion Algorithms: Performance Evaluation on a Shared Bus Architecture", Computers and Graphics, Vol.10, No.1, pp.7-25, 1986.

38. D. Ghosal and L.M. Patnaik, "SHAMP: An Experimental Multiprocessor System for Performance Evaluation of Parallel Algorithms", Multiprocessing and Microprogramming, Vol.19, No.3, pp.179-192, 1987.

39. K.P. Gostelow and R.E. Thomas, "Performance of a Data Flow Computer", Tech. Rep. 127a, Dept. Information and Computer Science, University of Califirnia, Irvine, CA, October 1979.

40. A.K. Goswami and L.M. Patnaik, "An Algebraic Approach Towards Formal Development of Functional Programs", accepted for publication, New Generation Computing.

41. A.K. Goswami, "Non-Determinism and Communication in Functional Programming Systems: A Study in Formal Program Development", Ph.D. Dissertation, Dept. Computer Science and Automation, Indian Institute of Science, Bangalore, India, October 1985.

42. A. Gottlieb, R. Grishman, C.P. Krushkal, K.P. McAaliffe, L. Randolph and M. Snir, "The NYU Ultracomputer -- Designing an MIMD Shared Memory Parallel Computer", IEEE Trans. on Computers, Vol.C-32, No.2, pp.175-189, February 1983.

43. Z. Halim, "A Data-Driven Machine for OR Parallel Evaluation of Logic Programs",

New Generation Computing, Vol.4, No.1, pp.5-33, 1986.

44. R.H. Halstead Jr., "Multilisp: A Language for Concurrent Symbolic Computation", ACM Trans. on Programming Languages and Systems, October 1985.

45. P.B. Hansen, "The Programming Language Concurrent Pascal", IEEE Trans. on Software Engineering, Vol.SE-1, No.2, pp.199-209, June 1975.

46. P.B. Hansen, "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, Vol.21, No.11, pp.934-941, November 1978.

47. J.P. Hayes, R. Jain, W.R. Martin, T.N. Mudge, L.R. Scott, K.G. Shin and Q.F. Stout, "Hypercube Computer Research at the University of Michigan", Proc. Second Conf. Hypercube Multiprocessors, September/October 1986.

48. W.D. Hillis, The Connection Machine, M.I.T. Press, Cambridge, MA, 1986.

49. C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", Communication of the ACM, Vol.17, No.10, pp.549-557, October 1974.

50. C.A.R. Hoare, "Communicating Sequential Processes", Communication of the ACM, Vol.21, No.8, pp.666-677, August 1978.

51. J.J. Hopfieed and D.W. Tank, "Computing with Neural Circuits", Science, pp.625-633, August 1986.

52. P. Hudak, "Para-Functional Programming", Computer, Vol.19, No.8, pp.60-70, August 1986.

53. K. Hwang and F.A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill Book Company, New York, NY, 1984.

54. A.K. Jones and E.F. Gehringer, "Cm* Multiprocessor Project: A Research Review", Tech. Rep. CMU-CS-80-131, Carnegie-Mellon University, July 1980.

55. Inmos Ltd., Occam Reference Manual, 1986.

56. Inmos Ltd., Transputer Reference Manual, Ref. No. 72TRN 04802, 1986.

57. R.M. Keller, S. Patil and G. Lindstrom, "A Loosely Coupled Applicative Multiprocessing System", Proc. Nat. Computer Conf., AFIPS, pp.861-870, 1979.

58. W.E. Kluge and H. Schlutter, "An Architecture for the Direct Execution of Reduction Languages", Proc. Int'l Workshop High-Level Language Computer Architecture, pp.174-180, May 1980.

59. R.A. Kowalski, "Algorithm = Logic + Control", Communication of the ACM, Vol.22, No.7, pp.424-435, July 1979.

60. D. Krishnan and L.M. Patnaik, "Systolic Architecture for Boolean Operations on Polygons and Polyhedra", Eurographics, The Europian Association for Computer Graphics, Vol.6, No.3, July 1987.

61. C.P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors", Proc. Int'l Conf. Parallel Processing, pp.236-240, August 1984.

62. H.T. Kung and S.W. Song, "A Systolic 2D Convolution Chip", Dept. Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-81-110, March 1981.

63. H.T. Kung, "Why Systolic Architectures?", Computer, Vol.15, No.1, pp.37-46, January 1982.

64. S.Y. Kung, S.C. Lo, S.N. Jean and J.N. Hwang, "Wavefront Array Processors -- Concept to Implementation", Computer, Vol.20, No.7, pp.18-33, July 1987.

65. R.P. Lippmann, "An Introduction to Computing with Neural Nets", IEEE ASSP, pp.4-20, April 1987.

66. G.A. Mago, "A Network of Multiprocessors to Execute Reduction Languages", Int'l J. Computer and Information Science, Part 1: Vol.8, No.5, pp.349-385, Part 2: Vol.8, No.6, pp.435-571, 1979.

67. P.C. Mathias and L.M. Patnaik, "A Systolic Evaluation of Linear, Quadratic and Cubic Expressions", accepted for publication, Journal of Parallel and Distributed Computing.

68. J. McCarthy, et.al., LISP 1.5 Programmer's Reference Manual, M.I.T. Press, Cambridge, MA, 1965.

69. C.A. Mead and L.A. Conway, Introduction to VLSI Systems, Addison Wesley, Reading, MA, 1980.

70. R. Onai, M. Aso, H. Shimizu, K. Masuda and A. Matsumoto, "Architecture of a Reduction Based Parallel Inference Machine: PIM-P", New Generation Computing, Vol.3, No.2, pp.197-228, 1985.

71. D.A. Padua, D.J. Kuck and D.L. Lawrrie, "High Speed Multiprocessor and Compilation Techniques", IEEE Trans. on Computers, Vol.C-29, No.9, pp.763-776, September 1980.

72. L.M. Patnaik, P. Bhattacharya and R. Ganesh, "DFL: A Data Flow Language", Computer Languages, Vol.9, No.2, pp.97-106, 1984.

73. L.M. Patnaik and B.R. Badrinath, "Implementation of CSP-S for Description of Distributed Algorithms", Computer Languages, Vol.9, No.3/4, pp.193-202, 1984.

74. L.M. Patnaik, R. Govindarajan and N.S. Ramadoss, "Design and Performance Evaluation of EXMAN: An EXtended MANchester Data Flow Computer", IEEE Trans. on Computers, Vol.C-35, No.3, pp.229-244, March 1986.

75. L.M. Patnaik and J. Basu, "Two Tools for Interprocess Communication in Distributed Data Flow Systems", The Computer Journal, Vol.29, No.6, pp.506-521, December 1986.

76. A. Plas, et.al., "LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment", Proc. 1976 Int'l Conf. Parallel Processing, pp.293-302, August 1976.

77. C.V. Ramamoorthy and H.F. Li, "Pipeline Architectures", Computing Surveys, Vol.9, No.1, pp.61-102, March 1977.

78. C.P. Ravikumar and L.M. Patnaik, "Parallel Placement Based on Simulated Annealing",

IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors, October 1987.

79. C.P. Ravikumar and L.M. Patnaik, "An Architecture for CSP and its Simulation", Int'l Conf. Parallel Processing, 1987.

80. C. Rieger, "ZMOB: Doing it in Parallel", Proc. Workshop Computer Architecture for PAIDM, pp.119-125, 1981.

81. A. Rosenfield, "Parallel Image Processing Using Cellular Arrays", Computer, Vol.16, No.1, pp.14-20, January 1983.

82. R.M. Russel, "The Cray-1 Computer System", Communications of the ACM, Vol.21, No.1, pp.63-72, January 1978.

83. C.L. Seitz, "The Cosmic Cube", Communication of the ACM, Vol.28, No.1, pp.22-33, January 1985.

84. E.Y. Shapiro, "Concurrent Prolog: A Progress Report", Computer, Vol.19, No.8, pp.44-58, August 1986.

85. L. Snyder, "Introduction to Configurable Highly Parallel Computer", Computer, Vol.15, No.1, pp.47-64, January 1982.

86. J.S. Squire and S.M. Palais, "Programming and Design Considerations for a Highly Parallel Computer", Proc. AFIP Conf., Vol.23, pp.395-400, 1963.

87. J. Silc and B. Robič, "Efficient Static Dataflow Architecture for Specialized Computatins", Proc. 12th IMACS World Congress on Scientific Computing, Paris, France, July 1988.

88. K.R. Traub, "A Compiler for M.I.T. Tagged-Token Data Flow Architecture", M.S. Thesis, M.I.T., Cambridge, MA, 1986.

89. P.C. Treleaven, "Principle Components of a Data Flow Computer", Proc. 1978 Euromicro Symp., pp.366-374, October 1978.

90. P.C. Treleaven and G.F. Mole, "A Multiprocessor Reduction Machine for User-Defined Reduction Languages", Proc. 7th Int'l Symp. Computer Architecture, pp.121-130, 1980.

91. P.C. Treleaven, D.R. Brownbridge and R.P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", Computing Surveys, Vol.14, No.1, pp.93-143, March 1982.

92. D.A. Turner, "A New Implementation Technique for Applicative Languages", Software Practice and Experience, Vol.9, No.5, pp.31-49, September 1979.

93. J.D. Ullman, Computational Aspect of VLSI, Computer Science Press, 1984.

94. S. Umeyama and K. Tamura, "Parallel Execution of Logic Programs", Proc. 10th Ann. Symp. Computer Architecture, pp.349-355, 1983.

95. S.R. Vehdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages", IEEE Trans. on Computers, Vol.C-33, No.12, pp.1050-1071, December 1984.

96. A.G. Venkataramana and L.M. Patnaik, "Design and Performance Evaluation of a Systolic Architecture for Hidden Surface Removal", accepted for publication, Computer and Graphics.

97. A.G. Venkataramana and L.M. Patnaik, "Systolic Architecture for B-Spline Surfaces", accepted for publication, International Journal of Parallel Programming.

98. I. Watson and J. Gurd, "A Prototype Data Flow Computer with Token Labelling", Proc. Nat. Computer Conf. New York, AFIPS Press, pp.623-628, 1979.

99. N. Wirth, "Modula: A Programming Language for Modular Multiprogramming", Software Practice and Experience, Vol.7, No.1, pp.3-35, January 1977.

100. M.J. Wise, "A Parallel PROLOG: The Construction of Data-Driven Model", ACM Conf. LISP and Functional Programming, 1982.

101. W.A. Wulf and C.G. Bell, "C.mmp -- A Multi-miniprocessor", Proc. AFIPS Fall Joint Computer Conf., Vol.41, pp.765-777, 1972.

102. W.A. Wulf, et.al., "Overview of the Hydra Operating System", Proc. 5th Symp. Operating System Principles, pp.122-131, November 1975.

103. R. Yang and H. Aiso, "P-Prolog: A Parallel Logic Language Based on Exclusive Relation", New Generation Computing, Vol.5, No.1, pp.79-95, 1987.

Dušan Surla
Prirodno-matematički fakultet
Novi Sad

UDK 519.7

ABSTRACT: The notion is introduced of the piercing point of a straight line ℓ through the hull of a simple polyhedron P. An intersection point of ℓ and the hull of P is considered to be the piercing point if the straight line at the intersection point passes from the interior to the exterior domain of P, or vice versa. It is proved that the point Z belongs to the interior domain of P if an arbitrary straight line ℓ, passing through Z, contains an odd number of piercing points on the located on the one side of Z. On the basis of this statement, an effective procedure is formed for determination whether a given point belongs to the interior domain of P. An algorithm based on this procedure, anables determination of those intervals on the straight line which belong to the interior domain of P.
Key words: Computational geometry, Point-Location, Polygon, Polyhedron.

ODNOS TAČKE I PROSTOG POLIEDRA. Uveden je pojam prodorne tačke prave ℓ kroz omotač prostog poliedra. Presečna tačka izmedju ℓ i omotača od P je prodorna ako prava u presečnoj tački prelazi iz spoljašnje u unutrašnju ili iz unutrašnje u spoljašnju oblast od P. Dokazano je tvrdjenje da tačka Z pripada unutrašnjoj oblasti od P, ako proizvoljna prava ℓ koja prolazi kroz Z sadr'i sa iste strane od Z neparan broj prodornih tačaka. Na bazi ovog tvrdjenja formirana je efektivna procedura za ispitivanja da li data tačka pripada unutrašnjoj oblasti od P. Koristeći ovu proceduru opisan je algoritam za izdvajanju intervala na pravoj koji pripada unutrašnjoj oblasti od P.

1. Introduction

One of the most fundamental problems in computational geometry is the so-called *point-location problem*. To the present, the attention has been mainly paid to location of a point in a planar subdivision. The problem can be stated as follows: Given a straight-line graph G having n vertices, and a point Z, determine what is the region of the planar subdivision induced by G in which Z is found. Main results on the study of this problem have been presented in [1-8]. Much less attention has been paid to solving the problem in three- and multi-dimensional space. In [9], the problem has been solved by generalization of the binary search, whereas in [10] the data structure has been described by a sequence of similar lists formed in such a way to ensure each list to be binary searched.

We consider first the following problem: Given a point Z and a simple polyhedron P, determine whether the point Z belongs to the interior domain of P. First, we shall describe the data structure to represent the polyhedron, and adopt a doubly-connected list. In addition, we shall present a procedure for separation of those edges which define a corresponding facet of P. Further, we describe in detail a new algorithm and the appropriate procedures by which it is possible to determine whether Z belongs to the interior domain of P.

2. Data Structure

The data structure adopted for representation of a polyhedron is of the form of a planar graph G=(V,E), where V and E are the sets of vertices and edges, respectively, of a polyhedron embeded in a plane. The graph G is represented as a doubly-connected edge list (DCEL) [11]. The main component of this graph is an edge node which contains 6 fields: V1, V2, F1, F2, P1, and P2. The fields V1 and V2 define the edges which are oriented from V1 to V2. The fields F1 and F2 contain the names of facets of P lying on the left, i.e. on the right, with respect to the edge orientation from V1 to V2. The field P1 (resp. P2) points out to the edge node which is encountered first if the edge (V1,V2) is rotated

couterclockwise around V1 (resp. V2).

Suppose that FACE (j,A,n) is the procedure for separation of the edges determining the j-th facet of P, where A_i, i=1,...,n, contains the addresses of the ordered edges which determine the j-th facet of P.

3. Description of the Algorithm

Denote the vertices, edges and facets of P by V_i, i=1,...,|V|; E_i, i=1,...,|E|; F_i, i=1,...,|F|, respectively. Denote with ℓ a half-straight line having the original in the point Z. The intersection point of ℓ and the hull of P is a *piercing* point if at this point ℓ passes from the interior to the exterior domain of P, or vice versa. Since ℓ is unlimited in one direction, there is a point on ℓ belonging to the exterior domain, such that all piercing points of ℓ through the hull of P and the query point Z are on the same side of that point. Starting from this point and going toward Z, at each odd piercing point ℓ enters the interior domain of P and each even point ℓ comes out of it. Therefore, the following statement holds: If the total number of piercing points of ℓ through the hull of P is odd, the query point Z belongs to the interior domain of P.

If ℓ lies in the plane of a facet, then that facet does not contain a piercing point. Let R be a crossing point of ℓ and a facet of P. If R belongs to the interior region of that facet, then R is the piercing point. If, however, R belongs to an edge, then it may be, or may be not a piercing point. In this case, an additional analysis would be required to determine whether R is a piercing point. Since ℓ is chosen in an arbitrary way, let it be such that the intersection of ℓ and E_i for i=1,...,|E| is an empty set.

Consider the set of all planes that are uniquely determined by the point Z and the edges E_i, i=1,...,|E|. Denote these planes by α_j, j=1,...,k, where $k \leq |E|$. Determine a new plane β which is different from all planes α_j, j=1,...,k, and passes through Z. Further, let form a set of half-straight lines H which lie in the plane β and all of them have their origin in Z, in the following way. First, if E_i lies in β then the half-straight line is determined by E_i and Z. This is an implication of the manner in which the planes α_j, j=1,...,k have been determined. Namely, if Z and E_i do not lie on the same straight line, then E_i and Z determine the plane α_j, j∈(1,...,k), which is in contradiction to the statement that α_j and β are different planes. Second, if the intersection of β and E_i is a pont, then the half-straight line is determined by that point and the point Z. In this way, a finite set of half-straight lines H in the plane β is formed. Since the set of half-straight lines (having their origin in Z) in the plane β is unlimited, it comes out that it is always possible to choose a half-straight line ℓ in β, originated in Z such that $\ell \notin H$. Therefore, the procedure for choosing ℓ can be formed in the following way.

```
procedure CHOOSE ( ℓ );
 begin
   k:=0;
   for i:=1 to |E| do
    if ( E_i and Z determine a plane ) then
      begin
        k:=k+1;
        α_k:=plane (E_i,Z);
      end;
   PLANE (β);
   (* This procedure determines plane β, so
      that Z ∈ β and β ≠ α_i, i=1,...k *)
   j:=0;
   for i:=1 to |E| do
     begin
      if ( E_i lies on β ) then
        begin
          j:=j+1;
          h_j:=half-straight line (E_j,Z);
        end;
       else if ( β ∩ E_i ≠ ∅ ) then
        begin
          R:=β ∩ E_i;
          j:=j+1;
          h_j:=half-straight line (R,Z);
        end;
     end; (*for*)
   HALFLINE ( ℓ )
   (* This procedure determines half-
      -straight line ℓ on β with the initial
      point Z and ℓ ≠ h_i, i=1,...,j *)
 end.
```

The half-straight line ℓ determined on the basis of this procedure is such that it has no common points with any edge of P. If the intersection of ℓ and the plane which is determined by a polyhedron facet is not an

empty set, then their intersection is a point. Now, it is necessary to determine whether this point belongs to the corresponding facet. Thus the task is reduced to determination of the relation between a point and simple polygon. The same statement holds as for the relation between a point and a polyhedron. The algorithm for determination whether a point belongs to the interior region of simple polygon consists in the following.

Let be given a simple polygon with ordered vertices V_i, i=1,...,n, and the point R. Denote with r an arbitrary half-straight line having the origin in R. The piercing point of r through the simple polygon can be determined in the following way. If an edge lies on r, then there are no piercing points on it. Suppose that r passes through the vertex V_i. If the neighbouring vertices V_{i-1} and V_{i+1} are on different sides of r, then V_i is a piercing point. If the condition
$((V_{i-1}-V_i)\times(R-V_{i-1}))\circ((V_{i+1}-V_i)\times(R-V_{i+1})) < 0$
is satisfied, then V_i is a piercing point, otherwise it is not. If r and an edge have one common point and that point is not a vertex, then it is a piercing point. In this way, the total number of piercing points can be determined. If this number is odd, then R belongs to the interior region of a simple polygon.

On the basis of the algorithm described, it is possible to form a procedure for determination whether there is a piercing point of ℓ through a facet of P. This procedure is of the form:

```
procedure FACEGON ( j,ℓ,A,n,test );
 begin
  if(plane(j) ∩ ℓ = ∅ ) then test:=false
  else
   begin
    R1:=plane(j) ∩ ℓ;
    R2:=V2[A[1]];
    A[n+1]:=A[1];
    k:=0;
    for i:=2 to n do
     begin
      j:=V1[A[i]];
      k:=V2[A[i]];
      Ej:=(Vj-R1)x(R2-Vj);
      Ek:=(Vk-R1)x(R2-Vk);
      if (Ej∘Ek<0) then
       begin
        k:=k+1;
        pr[k]:=edge(Vj,Vk) ∩
                    straight-line (R1,R2);
       end;
      else if(Ej∘Ek=0)
           then
            if(Ej=0) then if(Ek≠0) then
             begin
              m:=V1[A[i-1]];
              Em:=(Vm-R1)x(R2-Vm);
              if (Ek∘Em<0) then
               begin
                k:=k+1;
                pr[k]:=Vj;
               end;
             end;
            else if (Ek=0) then
             begin
              m=V2[A[i+1]];
              Em:=(Vm-R1)x(R2-Vm);
              if (Ej∘Em<0) then
               begin
                 k:=k+1;
                 pr[k]:=Vk;
               end;
             end;
     end; (*for*)
    SORT (pr,k,npr);
   (* npr is the number of different piercing
      points on the same side of R *)
    test:=false;
    (* ℓ is not piercing the
       j-th facet of P *)
    if(npr odd) then
     test:=true
     (* ℓ is piercing the j-th facet od P *)
   end;
 end.
```

On the basis of the statement given above and the introduced procedures, it is possible to form a routine for determining whether a given point belongs to the interior domain of a simple polyhedron. This procedure is of the form:

```
procedure TESTPZ ( test );
 begin
  pr:=0;
  CHOOSE (ℓ);
  for i:=1 to |F| do
   begin
    FACE (i,A,n);
    FACEGON (i,ℓ,A,n,test);
    if test then pr:=pr+1
   end;
```

```
test:=false (* Z does not belong to the
                interior domain of P *)
if ( pr odd ) then
   test:=true (* Z belongs to the interior
                domain of P *)
end.
```

By this procedure, the following task can be solved. Determine the intervals on a straight line which belong to the interior domain of a simple polyhedron. To solve this problem, the following alterations of the above procedure should be made. First, if the point R is a crossing point of this straight line and of an edge of a simple polyhedron, then an additional analysis should be carried out to determine whether this point is a piercing point. This can be achieved in the following way. Let choose two points which lie on the given straight line in an arbitrarily small vicinity of the point R, and that they are found on different sides of the point R. If one of these two points belongs to the interior and the other one to the exterior domain of the simple polyhedron, then R is a piercing point, otherwise it is not. Second, determine the piercing points and sort them out along the given straight line. These points determine the sequence of consecutive finite intervals, each odd of which belongs to the interior domain of P.

4. Conclusion

The procedures described enable the following: the choice of a half-straight line ℓ originated in the query point Z such that the intersection of ℓ and all the edges of simple polyhedron P is an empty set; determination whether ℓ pierces a facet of P. On the basis of these procedures a routine is formed to determine whether Z belongs to the interior domain of P. In addition, the other algorithm presented enables determination of the intervals on a given straight line which belong to the interior domain of P. Furthermore, the described procedures may be used for solving other problems in computational geometry (identification of hidden lines, determination and detection of intersections, etc.).

5. References

[1] Lee T. D., Preparata P. F., "Location of Point in a Planar Subdivision and its Applications", SIAM J. Comput. Vol. 6. No. 3, (1977) 594-606.

[2] Lee T D., Yang C. C., "Location of Multiple Points in a Planar Subdivision" Infor. Processing Lett., Vol. 9, No. 4, (1977) 190-193.

[3] Preparata P. F., "A Note on Locating a Set of Points in a Planar Subdivision", SIAM J. Comput. Vol. 8, No. 4, (1979) 542-545.

[4] Preparata P. F., "A New Approach to Planar Point Location", SIAM J. Comput. Vol. 10, No. 3, (1981) 473-482.

[5] Kirkpatrick D., "Optimal Search in Planar Subdivision", SIAM Comput. Vol. 12, No. 1, (1983) 28-35.

[6] Edahiro M., Kokubo I., Asano. T., "A New Point-Location Algorithm and its Practical Efficiency-Comparison With Existing Algorithms", ACM Trans. on Graphics, Vol. 3, No. 2, (1984) 86-109.

[7] Edelsbrunner H., Guibas L. J., Stolfi J., "Optimal Point Locatiom in a Monotone Subdivision", SIAM J. Comput. Vol.15, No. 2, (1986) 317-340.

[8] Sarnak N., Tarjan R., "Planar Point Location Using Persistent Search Trees", Com. ACM Vol. 29, No. 7, (1986) 669-679.

[9] Dopkin D., Lipton R., "Multidimension Searching Problems", SIAM J. Comput. Vol. 5, No. 2, (1976) 181-186.

[10] Cole R., "Searching and Storing Similar Lists", J. of Algorithms 7, (1986) 202-220.

[11] Preparata F., Shamos M., Computational Geometry, an Introduction, Springer - - Verlag 1985.

ODNOS TAČKE I PROSTOG POLIEDRA

dr Dušan Surla
Prirodno-matematički fakultet
Institut za matematiku
dr Ilije Đuričića 4
Novi Sad

Sadržaj Uveden je pojam prodorne tačke prave ℓ kroz omotač prostog poliedra. Presečna tačka izmedju ℓ i omotača od P je prodorna ako prava u presečnoj tački prelazi iz spoljašnje u unutrašnju ili iz unutrašnje u spoljašnju oblast od P. Dokazano je tvrdjenje da tačka Z pripada unutrašnjoj oblasti od P, ako proizvoljna prava ℓ koja prolazi kroz Z sadrži sa iste strane od Z neparan broj prodornih tačaka. Na bazi ovog tvrdjenja formirana je efektivna procedura za ispitivanja da li data tačka pripada unutrašnjoj oblasti od P. Koristeći ovu proceduru opisan je algoritam za izdvajanju intervala na pravoj koji pripada unutrašnjoj oblasti od P.

DERIVING PROTOCOLS FROM SERVICES IN THE FINITE STATE MACHINE REPRESENTATION

INFORMATICA 2/88

UDK 519.713

Monika Kapus-Kolar
Inst. Jožef Stefan, Ljubljana

A method is suggested for derivation of protocols from services, based entirely on the finite state machine representation. The method provides several suggestions for human intervention in the design process and thereby for a great variety of solutions. Other benefits are parametrization, transformations for data-flow optimization, uniform treatment of synchronous and asynchronous channels, a uniform approach to composition and decomposition of entities and thereby a uniform approach to design of services and protocols.

Izpeljava protokolov iz servisov ob uporabi predstavitve s končnimi avtomati. Predstavljena je metoda za avtomatsko konstrukcijo komunikacijskih protokolov za realizacijo podanega globalnega servisa, ki v celoti temelji na predstavitvi s končnimi avtomati. Metoda je zelo primerna za interaktivno delo, ki vodi v široko paleto rešitev. Druge dobre lastnosti metode so parametrizacija, transformacije za optimizacijo pretoka podatkov, enotna obravnava sinhronih in asinhronih kanalov in enoten pristop h kompoziciji in dekompoziciji osebkov, ki omogoča poenotenje načrtovanja servisov in protokolov.

0. Introduction

Derivation of a communication protocol from a given service specification is one of the most challenging problems in the field of computer networks. Two methods have been proposed so far, which we find particularly interesting, because they provide algorithms for totally automatic construction of a suitable protocol. The method, proposed in [Prin], constructs a Petri-net type protocol specification from a finite-state machine service specification, while the method of [BochGotz] is based on attribute grammars. In our paper, we are proposing a method, based entirely on finite state machines, which follows the selection / resolution principle and is therefore also suitable for construction of protocols in a man-machine dialogue.

We assume that a distributed system consists of a set of entities, communicating with each other and the environment through a set of reliable two-point channels. Some of the channels are unbounded FIFOs with unknown delays (asynchronous), while the others are synchronous (the "rendez-vous" type of communication).

A global service, which a system should provide to the environment, is specified by a finite state machine **G**, with edges representing an asynchronous transmission or reception of a particular message by the system on a particular external channel or a synchronous external event. Paths, leading from the starting state of **G**, represent the characteristic sequences of system actions.

A message is a tuple of parameters, posessing explicitely or implicitely stated identifiers and values. The crucial observation about parameters is that each parameter identifier, occurring in a specification, represents a global system variable, which is concurrently updated or read by the entities. Parameters, exchanged in an action, are its **output parameters**, while parameters, generating the values of the output parameters, are the **input parameters** of the action. If an action is not an asynchronous transmission, the values of its parameters can also be obtained from the environment. **G** must possess the following properties:

Property 0.1: It must not contain two non-terminal states S_1 and S_2, such that for every outgoing edge **a** in S_1, leading to a state **S**, there is also the same outgoing edge in S_2, and vice-versa (equivalent states).

Property 0.2: If in a state S_1, there are two paths P_1 and P_2, such that the action sequence of P_2 is a permutation of the action sequence of P_1, respecting all causality relations of P_1 (P_2 is equivalent to P_1), they must both lead to the same state S_2. Path equivalence is formalized in the section 1.

Property 0.3: If there is an edge, representing an action, requiring a value of a particular parameter as an input, then the edge must be preceded from any direction by some edge, generating the value.

The first step in our protocol design method is to convert **G** into another finite state machine G_P, which mirrors particular design decisions about parallel execution of external actions of a system, while respecting the causality relations of **G**. Then the states of G_P, requiring communication between entities, are identified. For each such state, a procedure for exchanging messages on internal channels must be provided by a designer. Such procedures explicitate externally invisible transitions of a system, which are initially hidden in the states of **G**, as indicated in G_C,

an extended version of G_P. Note also that it is execution of the internal procedures, that entities without access to external channels are used for. At that point it may turn out that the task can not be solved with the existing channels. By integrating the internal procedures into G_C, another global system behaviour specification G_I is obtained, which is data-flow optimized into G_O and subsequently used for generation of finite state machines for individual entities.

Two algorithms will be used extensively throughout the paper: **Algorithm 0.1**, which introduces to a state machine a new path, and **Algorithm 0.2**, which deletes a particular path.

Algorithm 0.1:

{create a new path P}

```
begin{Algorithm 0.1}
 represent P by a finite set of finite
  segments
 (not all types of the representation might be
  suitable for a particular purpose);
 for every segment, which is a concatenation
     of an action sequence α and an action
     a and should lead from S1 to S2 do
  begin
   if there is no state Sk, different from S1,
      with a single outgoing edge, namely
      a, leading to S2, or with a single
      incoming path, namely α, with the
      initial state S1
   then create Sk as a new state;
   if in Sk, there is no outgoing edge a
   then create an edge a from Sk to S2
   else
    if the edge a leads to a state, different
       from S2,
    then exit with error;
   if in S1, there is no outgoing path α
   then create a path α from S1 to Sk;
   else
    if the path α leads to a state, different
       from Sk,
    then exit with error;
   merge the equivalent states
  end
end{Algorithm 0.1}.
```

Algorithm 0.2:

{delete a particular path, leading from S_1 to S_2}

```
begin{Algorithm 0.2}
 for every state S of the path do
  begin
   if in S, there are some incoming edges, not
      lying on the path, and also some, lying
      on the path,
   then
    begin
     create a state Se, equivalent to S;
     redirect the incoming edges of S, not
       lying on the path, to Se
    end;
 for every edge e of the path do
  if all outgoing edges of the destination
     state of e are lying on the path
  then delete e;
 delete the unconnected states;
 merge the equivalent states
end{Algorithm 0.2}.
```

1. Converting a Global Service Specification into an Equivalent Form with a Desired Degree of Parallelism

Service specifications in [BochGotz] use three types of composition (parallel, sequential and alternative). This versatility makes it difficult to identify actions, which could be enabled concurrently, as in a specification, they might lie far apart.

In a finite state machine specification, parallel composition of actions is represented by various permutations of the actions, connecting the same pair of states, with parameters inducing no causality relationship between the actions. The desired degree of parallelism in a global service specification can be achieved by repeated application of **Transformation 1.1**, which increases parallelism, and **Transformation 1.2**, which decreases it.

The idea behind **Transformation 1.1** is that if there is a path P from a state S_1 to a state S_2, the two states may also be connected by all paths, equivalent to P. P_1 **is equivalent to** P_2, iff there is a path P, such that the action sequences of P_1 and P_2 can be generated from P by zero or more applications of **Transformation 1.1**. **Transformation 1.1** generates equivalent paths by repeatedly selecting an action a_2 of the current action sequence and moving it towards the start of the sequence. If in that process a_2 meets an action a_1, such that a_1 is a potential necessary condition for a_2 (**Predicate 1.1**), a_2 may not move any further. We use the word "potential", because G might negate the causality relationship between two actions by providing an alternative path with the two actions in the reverse order.

Predicate 1.1:

(a_1 is a **potential necessary condition** for a_2)

```
begin{Predicate 1.1}
 Predicate 1.1:=
  (a1 is a synchronous action or a reception
   and
   a2 is a synchronous action or a trans-
    mission)
  or
   a1 and a2 are two actions on the same
    channel
  or
   a1 generates a parameter value, which is
    read or redefined by a2
end{Predicate 1.1}.
```

Transformation 1.1: If there is a path $a_1 a_2$, connecting S_1 and S_2, and a_1 is not a potential necessary condition for a_2 (**Predicate 1.1**), then it is possible to create (by **Algorithm 0.1**) a path $a_2 a_1$ from S_1 to S_2.

To achieve the highest possible degree of parallelism, **Transformation 1.1** should be applied as long as possible. On the other hand, we want G_P to be a **finite** state machine, but if there is a cycle C and an action a, such that it can move through the cycle for ever (as no action of the cycle is a potential necessary condition for it) and C contains at least two different actions, the set of paths, equivalent to the cycle, is infinite and **can not be described by a finite state machine**. Therefore **Transformation 1.1** must be applied under designer's control.

If a_1 is not a potential necessary condition for a_2, a system is free to execute the actions in the reverse order, because the environment **can not observe** it. **Transformation 1.1** adds a path, which represents execution of the actions in the reverse order, but as the environment can not observe the existence of such a path, a designer is also free to delete it from G by **Transformation 1.2**.

Transformation 1.2: If there is a path $a_1 a_2$ from S_1 to S_2 and a path $a_2 a_1$ from S_1 to S_2 and a_2 is not a potential necessary condition for a_1 (**Predicate 1.1**), then it is possible (by

Algorithm 0.2) to delete the path $a_1 a_2$.

2. Identifying States, Which Require Internal Communication

Some states in G_P might require communication between entities. In this section we introduce **Algorithm 2.2**, which generates another global system specification G_C by extending G_P with internal communication requirements.

Observing the actions, possible in a given state **S**, some of them may be enabled simultaneously and some not. Simply speaking, a set of actions may be enabled simultaneously, if they are in parallel or in exclusive composition. The next design step is to identify in each state **S** a set of exclusive compositions of parallel compositions of multisets of actions, possible in **S**, which might be selected by a system for simultaneous enabling. **Algorithm 2.1**, if not effected by designer's decisions, generates a solution with the highest possible degree of parallelism and minimal amount of internal communication, securing complete implementation of a service. The algorithm should be called systematically from **Algorithm 2.2**.

Algorithm 2.1:

{**Al** : the set of all **alternatives** of a given state **S**}
{Al_a : the set of **groups of alternatives**, which may be selected for simultaneous enabling in **S**}

begin{Algorithm 2.1}
 find **A**, the set of all actions possible in **S**;
 find **Al**, the set of all non-empty multisets of actions in **A**, such that the members of each multiset are in parallel composition and lead to a final state or a state with an outgoing edge, labeled by an action **a**, which is not in parallel composition with the members of the multiset or must not be added to the multiset because of a designer's decision
 {members of a multiset are in parallel composition, iff they may access their parameters simultaneously, each permutation of them is represented by an outgoing path in **S** and any two prefixes of the paths with the same multiset of actions lead to the same state};
 find Al_a, the set of all subsets of **Al**, which are maximal in respect to the following property **P** (for special control purposes, a designer may also decide to cover **Al** with subsets, which do poses the property **P**, but are not maximal):
 {a subset X of a set Y is maximal in respect to a property **P**, iff it has the property **P**, but can not be extended by any other members of Y without loosing the property}
 if all members (alternatives) of a member X of **Al** are enabled simultaneously, the entities, participating in their execution, are always able to select one of the alternatives without any internal communication
 (the global decision is equivalent to a set of local decisions)
end{Algorithm 2.1}.

Al in **Algorithm 2.1** answers the question, which actions may be enabled simultaneously, because they are in parallel composition, but one has to be careful. First, although we wish to enable simultaneously as many actions as possible, strict application of that rule might lead to an incomplete implementation of a service. Second, if in a state **S**, there is a loop with all edges labeled with the same label, it is possible to define an infinite **Al**, which requires careful definition of Al_a and careful construction of paths in **Algorithm 2.2**.

The idea behind grouping of alternatives is that a global decision procedure for selecting an alternative for actual execution might to some extent be performed as a set of local decision procedures. Respecting the property minimizes the amount of internal communication and at the same time provides a solution to the problem that actions for further execution can only be discussed among entities in terms of their a priori properties (as the only a priori property of a reception is its channel, it might not be possible to distinguish between two alternatives).

If only a partial implementation of a service is required, **Algorithm 2.1** is the most suitable point for human intervention. Partial implementations can be generated by definition of incomplete sets of alternatives or groups of alternatives.

Algorithm 2.2:

begin{Algorithm 2.2}
 Open:= [starting state of G_P];
 Closed:= [];
 G_C is just the starting state of G_P;
 while not Open=[] do
 begin
 move a state S_D from **Open** to **Closed**;
 find (by **Algorithm 2.1**) **Al**(S_D) and Al_a(S_D);
 for each member A_a of Al_a(S_D) **do**
 begin
 if A_a is not the only member of Al_a(S_D) or special guarding is required
 then add to G_C a τ_D edge from S_D to a new state S_I
 (a state is new, iff there is no state with the same name neither in G_P nor in G_C)
 else S_I:= S_D;
 find I(A_a), the set of input parameters of A_a;
 if I(A_a) is not empty
 then
 begin
 for each member **In** of I(A_a) **do**
 begin
 find **U(In)**, the set of entities, using the value of **In** in execution of A_a;
 find **K(In)**, the set of entities, knowing the value of **In**
 end;
 create an edge τ_P from S_I to a new state S_A
 end
 else S_A:= S_I;
 {create in S_A a graph G_A, representing execution of A_a:}
 for each outgoing path of S_D in G_P, representing execution of one of the members of A_a, **do**
 create the same outgoing path in S_A in G_C
 {Add as few new edges as possible (**Algorithm 0.1**), but keep paths, belonging to different groups of alternatives, disjoint. Do not use in G_A any old state names.};
 find **Pr**(A_a), the set of all entities, participating in execution of A_a;
 for each member **E** of **Pr**(A_a) **do**
 select **T(E)**, the set of all action sequences with a length>=0, executed by **E** as part of execution of A_a, after which **E** might decide to abandon execution of A_a and enter a synchronization procedure
 {although **T(E)** is selected by a designer, it has some mandatory members: the sequences, after which **E** has no asynchronous transmission to execute in A_a};

```
    find T, the set of all synchronization
     states of GA
    {S is a synchronization state of GA, iff
     for every entity E, the projection of a
     path from SA to S on the actions of E is
     in T(E)};
    find TN, a version of T, in which every
     member is replaced by its old name (the
     name of the equivalent state in GP);
    for each member SN of TN do
     begin
      if SN is not yet in GC
      then add SN to GC;
      if not SN in Closed
      then add SN to Open
     end;
    for each member SB of T do
     begin
      find its old name SN;
      create in GC a τB edge from SB to SN
     end
   end
  end;
 terminal states of GC:= terminal states of GP
end{Algorithm 2.2}.
```

Each edge τ_P requires execution of a **parameter distribution procedure**.

Each state S_D, coming onto **Open** in **Algorithm 2.2**, requires a global decision, what to do next, and is therefore called a **decision state**. If in S_D, there are several groups of alternatives or special guarding is necessary, then S_D requires execution of a **decision procedure**. In G_I, decision procedures are represented as trees of τ_D edges in decision states (S_D).

After a group of alternatives A_G has been selected and enabled, it starts executing. After executing A_G for some time, control of the participating entities is gradually transferred to a **synchronization procedure**. States of G_A, in which **all** the entities might enter a synchronization procedure, are called **synchronization states**. In G_C, synchronization procedures are represented as τ_S edges in synchronization states (S_S). With the help of a synchronization procedure, a system synchronizes to a state S_N of G_P, which corresponds to the currently active synchronization state.

The aim of firing a synchronization procedure after successful execution of one of the enabled alternatives is distribution of the knowledge that the actions, guarded by the alternative, are now enabled. The aim of firing a synchronization procedure before successful execution of any of the enabled alternatives is resynchronization of a system, after which another group of alternatives may be selected. This might be necessary, if the environment is not forcing the same group of alternatives as the system and does not cooperate promptly.

To minimize the amount of internal communication, an entity should fire a synchronization procedure only when it has no other action to execute without cooperation of the environment, but in principle, a designer might also define some additional synchronization states. When entering a synchronization procedure, the entity does not know, which of the enabled actions have already been executed by other entities. Therefore definition of synchronization states should be consistent, as stated in **Algorithm 2.2**.

If in a decision state, there are several groups of alternatives and a system is trying to execute one of them by repeatedly selecting a group, trying for some time to execute it and (if not successful) resynchronizing, some actions are enabled infinitely often, but not all the time. If the pending actions are synchronous, this is a degradation of fairness of the system, which is due to a particular distribution of external channels among the entities.

3. A General Design Method for Internal Procedures

The next task is to construct a finite state machine G_D by integrating into G_C the necessary internal procedures. G_D should represent the total behaviour of a system in a concise style, similar to that of G_P.

In G_P, all actions are on external channels, which have two end-points, but are observed only from the side of the system, while the actions, constituting internal procedures, are on internal channels with both end-points within the system. An action on an asynchronous internal channel actually consists of two events: transmission of a message and reception of the message. To retain the specification style of G_P, all actions should be represented in G_D as single events and their granularity should not become apparent before projecting G_D onto individual entities.

Let's ignore for a moment the external actions of a system and concentrate on its internal actions - the protocol. We argue that **a general purpose protocol should be specified by a single deterministic finite state machine P, representing the characteristic sequences of message transmissions and synchronous events.** In that way, a designer is forced to concentrate entirely on inter-entity causality relations of the protocol and not to rely upon intra-entity causality relations, which should be treated as **implementation details**. The approach is a direct application of the "empty medium abstraction" heuristic, which has proved to be useful for protocol verification, to protocol synthesis.

Specifications of individual entities can be generated from a global protocol specification P by **Algorithm 3.1** and **Transformations 1.1** and **1.2**. **Algorithm 3.1** projects P on one of the entities E, so that all actions on its incoming channels become receptions. Then **Transformations 1.1** and **1.2** are applied to specifications of individual entities to obtain the desired degree of intra-entity parallelism. In the two transformations, E represents a system, and the entities, cooperating with it, represent its environment.

Algorithm 3.1:

{projecting a global protocol specification P onto an individual entity E}

```
begin{Algorithm 3.1}
 while applicable do
  begin
   if there is an edge from S1 to S2, labeled
      by an action on a channel, which is not
      connected to E
      or
      there is an edge a from S to S1 and an
      edge a from S to S2
   then merge S1 and S2 into a single state;
   if there are two or more a edges from S1 to
      S2
   then replace them by a single a edge
  end
 end{Algorithm 3.1}.
```

Application of **Transformations 1.1** and **1.2** might result in several different sets of individual entity specifications. But this ambiguity of a global protocol specification P is not a deficiency of the specification method: As delays of all asynchronous channels are totally unknown, the sets can not be

distinguished by observing the entities for a finite period of time, hence the ambiguity is immaterial and any attempt to remove it (by explicitly mentioning asynchronous receptions in the global state machine or by specifying the protocol by a set of local state machines) is an **overspecification** and should be avoided.

The basic problem in protocol synthesis is to avoid deadlocks, unspecified receptions and unspecified parameters. When designing a global protocol specification of our type, those design errors can be avoided by respecting five simple common sense **Rules 3.1** to **3.5**.

Considering only the basic semantics of a state machine, each node represents an exclusive composition of the outgoing paths, but in protocol specification, there is also another, equally important type of composition - the parallel composition of actions. Parallel composition of actions can be described by exclusive composition of their permutations, but this mental task is not trivial enough to be carried out subconsciously. A potential deadlock or an unspecified reception occurs whenever some actions are in parallel composition by the nature of the system architecture, but that fact is not properly described by a state machine, usually because a designer is not aware of the existence of the parallel composition.

Rules 3.1 and **3.2** define paths, which must mandatory be specified, while **Rules 3.3** to **3.5** define some mandatory properties of the specified paths.

Rule 3.1: If **A** is a subset of actions, which are labels of the outgoing edges of a state **S**, such that every entity participates in execution of at most one member of **A** (an asynchronous transmission has one participant, the sender, and a synchronous action has two participants) - the actions are in parallel composition, then every permutation of the members of **A** must be represented by an outgoing path of **S**, as no entity is allowed to make any assumptions about execution of the actions of other entities, which it is not guarding. In the case of parametrization, any two actions, possible in a state **S**, on different channels, which are not both synchronous, must also be considered as in parallel composition and obey **Rule 3.1**, although the actions share a participant. This is to guarantee the soundness of **Rule 3.5**.

Rule 3.2: If in a state **S**, there is an outgoing path a_1a_2 and, by **Rule 3.1**, an outgoing edge a_2 must not be created in **S** without creating an outgoing path a_2a_1, then the path must actually exist.

Rule 3.3: If in a state **S**, there are two outgoing paths with the same multiset of actions **M**, such that no two different members of **M** belong to the same channel and no two different synchronous members of **M** share both participants, then the two paths must lead to the same state, as no entity can communicate to the rest of the system any information about the order, in which it has executed the actions of **M**.

Rule 3.4: Projection onto any entity must have **Property 0.3**.

Rule 3.5: If two actions are in parallel composition and one of them is generating a value of a parameter, then the other must neither read nor redefine the value.

Formal proof of the rules if outside the scope of the paper. Intuitively, they prevent unspecified receptions, because receptions are hidden in transmissions, they prevent deadlocks, because there is no state without transmissions and they guarantee coordinated progress of all participating entities, because any assumptions about non-existing information exchanges are avoided.

Returning to our original task, we point out that the initial service specification **G** for a system **S** under design should be obtained by the same method. **S** should be considered as an entity of a wider **closed** system **W**, consisting of **S** and the relevant entities, external to **S**. A designer should first specify a "protocol" for the system **W**, so that he is forced to think about implications of communication on the channels, connecting entities, external to **S**, on the service requirements for **S**. Then **G** can be generated by **Algorithm 3.1**.

As suggested in the section 4, the method should also be used for design of internal procedures, introduced by G_c.

4. Design of Parameter Distribution, Decision And Synchronization Procedures

In the section 2, we have defined three types of internal procedures: parameter distribution procedures, decision procedures and synchronization procedures. The nature of a protocol is mainly determined by decision procedures, while procedures of the other two types only play an auxiliary role. In our method, design of internal procedures and their integration is guided by eight basic heuristics:

Heuristic 4.1: Initially, each internal procedure should appear in the specification separated from the others. Message merging is subject to the final optimization (section 5).

Heuristic 4.2: An internal procedure should initially be scheduled just before its results are necessary. Earlier scheduling is subject to the final optimization.

In particular, parameter distribution procedures are inserted in G_c instead of τ_P edges. Decision procedures are inserted in G_c instead of τ_D trees, so that the starting state of a procedure is located at the root and its terminal states at the leaves of a tree. For synchronization procedures, the simplest kind of their integration into G_c is a bit more complicated and will be discussed later. The place for their integration is indicated by τ_s edges.

Heuristic 4.3: To prevent harmful re-ordering of messages, belonging to various internal procedures, during their transport, all participants of an internal procedure must agree on its termination, so that the internal procedures can be treated as atomic. Note that this is a general solution to the problem, described in the section 3.3 of [BochGotz]. If some of the messages are redundant, they can be deleted in the final optimization, which might sometimes result in the solution from [BochGotz].

Heuristic 4.4: The main point in design of an internal procedure is to determine for each of its terminal states **T** the **synchronization set** **Sy(T)**, the set of all entities, which must know that the system will progress through **T**. As at that point of design, internal procedures are scheduled just in time, the members of a synchronization set **Sy(T)** are exactly the entities, executing the actions, possible in **T**. When the participants of an internal procedure have reached an agreement on its termination (which is in a terminal state **T**), the members of **Sy(T)** must know, that the execution has terminated in **T**.

Heuristic 4.5: As the basic aim of an internal procedure is to lead a system to a particular state, it should be designed as an exchange of proposals about the terminal state, which the procedure should reach, and sets of terminal states, suggested by various participants, should be explicitly visible in the messages, so that the terminal state, which a path is leading to, can be calculated as an intersection of the sets, exchanged along the path. Beside that, terminal states **must** appear in the messages with the same names as in G_c. If the requirements are too rigorous, they can be overcome in the final optimization.

Heuristic 4.6: If an internal procedure is a parameter distribution procedure, it must communicate the necessary parameter values from the members of the relevant K sets to the members of the relevant U sets (see **Algorithm 2.2**).

Heuristic 4.7: We require that internal procedures are provided by a designer (in the spirit of the section 3), but this is not a serious drawback for the automatization of the protocol design process, as in practice, decision procedures, and even more procedures of the other two types, are drawn from a small set of types, which can be pre-constructed and used with suitable parameters, whenever necessary. An internal procedure must respect **Rules 3.1** to **3.5**, where **Rule 3.4** must be checked in regard to the rest of the system specification.

Internal procedures can not be designed in an optional order. The algorithm is the following:

1. Determine synchronization sets of parameter distribution procedures and design the procedures.
2. Determine synchronization sets of decision procedures and design the procedures.
3. Determine synchronization sets of synchronization procedures and design the procedures.

Now we are ready to define an algorithm for integrating into G_c a synchronization procedure. Observing a graph G_A, generated by **Algorithm 2.2**, it is not sufficient to replace by some procedures the τ_s edges in its synchronization states. The whole G_A, together with its τ_s edges, must be replaced by a graph G_s (the starting state of G_s is the starting state of G_A, the terminal states of G_s are those, pointed to by τ_s edges), concisely representing the action sequences of the expression:

$$\|_{E \in Pr(A_E)} (+_{p \in T(E)} (s.P(s)))$$

The expression has the following meaning: For each member **s** of a **T(E)**, design an internal procedure **P(s)**, put **s** and **P(s)** into sequential composition, put the expressions, belonging to various members of **T(E)**, into **or** composition, then put the expressions, belonging to various member of **Pr(A_E)**, into parallel composition.

With other words: each entity **E**, participating in execution of an A_E, executes an action sequence **s**, mandatory followed by a procedure **P(s)**, which distributes the knowledge of **E** about **N**, the set of the possible terminal states of G_s, as known by **E** after execution of **s**, to the members of the union of the synchronization sets of those states. **M** *is a member of* **N**, iff in G_A, there is a synchronization state **S**, connected with **M** by a τ_s edge, reachable from the starting state of G_A by a path, whose projection onto **E** is **s**.

The terminal state **T**, to which a path of G_s should lead, can be determined from the path by **Heuristic 4.5**. The requirements of **Heuristics 4.3** and **4.4** must be fulfilled on G_s as a whole. It turns out, that it is sufficient to fulfil **Heuristic 4.4** for each **P(s)**, but for **Heuristic 4.3** that might not be true. Hence, it is necessary to "blow" each terminal state **T** of G_s into a termination agreement procedure for all entities, participating in G_s. Procedures in all terminal states of G_s must be the same.

The principles, used in the design of G_s, lead to another heuristic for construction of internal procedures:

Heuristic 4.8: The first step in design of an internal procedure is to identify the knowledge, which is to be communicated. For each piece of knowledge (which might be a parameter value or a set of suggested terminal states), construct a procedure, which conveys the knowledge from its source to its destination. Put all such procedures into parallel composition and finally put the resulting procedure into sequential composition with a termination agreement procedure for all potential participants.

The result of the integration of internal procedures into G_c is a finite state machine, which might have some equivalent states, that have to be merged. Beside that, it might be necessary to introduce some new paths, required by **Rules 3.1** and **3.2**. As shown in the section 5, the resulting machine G_I is further optimized into G_o.

5. Final Optimization of a Global Service Provider Specification

Final optimization is performed by application of **Transformations 5.1** to **5.4**. The transformations address **Rules 3.1** to **3.5**, which use the notion of an action participant. The external actions of a system (those from the initial service specification) must be treated as internal actions of particular entities, which are their only participants. The transformations may only be applied, if they do not change the order of external actions.

Transformation 5.1: If **Rules 3.3** to **3.5** are not violated, then it is possible to introduce (by **Algorithm 0.1**) a particular path and all paths, required by **Rules 3.1** and **3.2**.

The transformation could be used for increasing parallelism or for moving scheduling points of internal procedures.

Transformation 5.2: Let **O** be the set of outgoing edges of a state **S**. Identify **P(O)**, the set of all paths, mandatory in **S** by **Rule 3.1**. Suppose that a member **a** of **O** is removed from **S**. Identify **P(O\a)**. If **Rules 3.4** and **3.5** are not violated, then it is possible to delete (by **Algorithm 0.2**) the members of **P(O)** and introduce to **S** (by **Algorithm 0.1**) the members of **P(O\a)** and all paths, required by **Rules 3.1** and **3.2**.

The transformation could be used for decreasing parallelism or for deleting redundant actions.

Transformation 5.3: If **Rules 3.1** to **3.5** are not violated, it is possible to apply a particular change of edge labels and merge the resulting equivalent states and edges.

The transformation could be used for decreasing the number of message types or for the final naming of messages.

Transformation 5.4: If **Rules 3.3** to **3.5** are not violated, then it is possible to replace (by **Algorithms 0.1** and **0.2**) a path between a pair of states by another path between the same pair of states and then add (by **Algorithm 0.1**) all

paths, required by **Rules 3.1** and **3.2**.

The transformation could be used for changing the order of actions or for merging of actions (messages).

Whenever possible, the transformations should be applied to such parts of a finite state machine, that **Rules 3.1** and **3.2** do not induce any new paths or their destination states are determined by **Rule 3.3**. For instance, if an action is executed without knowing, if it will be necessary at all (optimistic scheduling, introduced e.g. by **Transformation 5.1**), the destination state of a new path p_1, which includes an unnecessary execution of the action, must be provided by a designer. The suggested heuristic is to direct p_1 to the same state as p_2, which consists of the same sequence of actions as p_1, except that the unnecessary action is deleted.

6. Conclusions

In comparison to [BochGotz], which generates an unique solution, our method provides several suggestions for human intervention in the design process and thereby for a greater variety of solutions. Other benefits are parametrization, transformations for data-flow optimization, uniform treatment of synchronous and asynchronous channels, a uniform approach to composition and decomposition of entities and thereby a uniform approach to design of services and protocols. Similar conclusions can be drawn when comparing our method to [Prin].

If necessary, the design process can be fully automatized. The only condition is existence of parametrized transport procedures and termination agreement procedures and of some rules, which prevent the process from construction of infinite machines.

References

[Prin] Prinoth.R.: "An Algorithm to Construct Distributed Systems from State-Machines", in Sunshine.C.(ed.): "Protocol Specification, Testing, and Verification", pp.261-282, North-Holland, 1982

[BochGotz] Bochman.G.v., Gotzhein.R.: "Deriving Protocol Specifications from Service Specifications", Proceedings of the ACM SIGCOM Symposium, pp. 148-156, 1986

A. Milton Jenkins
Graduate School of Business
Indiana University

John V. Carlis
Department of Computer Science
University of Minnesota

UDK 681.326

Keywords and Phrases: Control Flowcharting, System Flowcharting, Dual Charts, Data Driven, Control Framework.

CR Categories: 1.3, 2.2, 2.43, 4.0

ABSTRACT

This paper describes an alternative approach to system flowcharting based on the use of dual charts -- one illustrating data flows, the other controls. The use of both system flowcharts and control flowcharts enhance the communicative power of systems documentation.

A specific system flowcharting technique is described. This type of system flowchart is the datum for the construction of a control flowchart. A control framework is developed and a conceptual model presented to provide a perspective for the novice system designer. Guidelines are given for the development of a control flowchart.

INTRODUCTION

A control flowchart is a document generated during the design phase of the system development life cycle. The control flowcharting technique described in this paper is predicated on two other system analysis and design concepts. The first, a system flowcharting technique, described in section two of this paper, is Chapin's "sandwich" concept utilizing ANSI Standards [4]. The second, a two dimensional system control framework, described in section three, is based on Gordon B. Davis's Data Flow Model (Computer Data Processing [6]). These two concepts are mutually supportive, and each has been utilized (in various forms) both in practice and as an educational device.

Controls are often specified on the system flowchart, producing a complex, cluttered, and sometimes confusing document (see Figure 1.) Yet the increasing emphasis on controls and the increasing complexity of systems (including user-developed systems) makes the need for clear documentation increasingly important. Toward this end, in 1971, Robert I. Benjamin [3] suggested an alternative approach in flowcharting; using dual charts -- one illustrating data flows and the other depicting controls, but nothing has since appeared in the literature developing this basic concept. Section four presents Benjamin's concept and develops a detailed technique for control flowcharting. (The

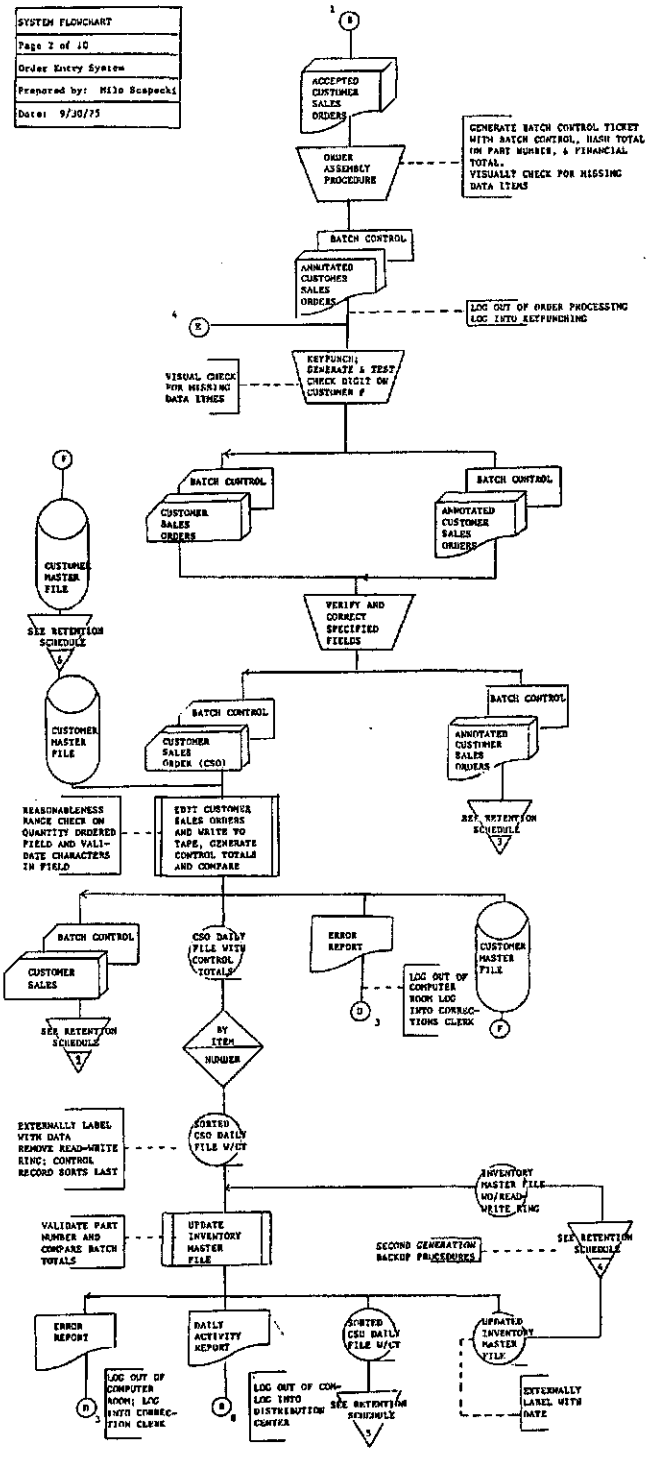


Figure 1

technique is easily integrated into life cycle methodologies and can be quickly used by both systems professionals and end-users. The basic objective of the systems control flowchart is to show clearly what controls exist and where they exist in a given system.

SYSTEMS FLOWCHARTING TECHNIQUE

There are two major and different approaches to system flowcharting. The first, and most widely used in the United States, is linked to program flowcharting both in the symbology utilized and in the logic-flow convention -- top to bottom and left to right and is often referred to in the literature as "computer" flowcharting. The second has its origins in the systems and industrial and mechanical engineering fields, and illustrates work flows, document flows, and process and procedural flows [1 & 12]. Generally it utilizes a version of the A.S.M.E. (American Society of Mechanical Engineers) symbols and a horizontal (left to right) logic-flow convention.

The technique presented here is a specific version of the "computer" flowcharting. As described by Chapin in his tutorial [4], it has two major facets; the sandwich concept and the use of American National Standard Flowchart Symbols [2].

The Sandwich Concept

Chapin describes the basic format of the sandwich rule as a flowchart composed of alternating layers of data identifications [the bread] and process identifications [the filling] as illustrated in Figure 2. The system flowchart is composed of a series of similar relationships where the output of one process may serve as the input to a subsequent process, thus representing the data-flow through the system. The system flowchart always begins and ends with symbols representing data.

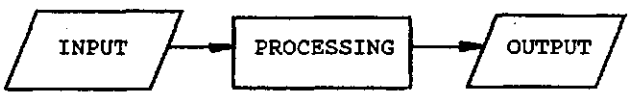


FIGURE 2: THE SANDWICH CONCEPT

This technique enhances the communicative power of the system flowchart. It is most effective in flowcharting a data-driven system -- a system where data physically exists on media outside the process, e.g., on magnetic tape, disks, documents, etc. throughout the system. Most traditional data processing systems and many user-developed systems fall into this category. The technique becomes more difficult to use in an event-driven (program-driven) system, e.g., where the data is temporarily stored in core within a process or between programs. In this situation the level (macro/micro) of system detail desired will influence the effectiveness of the sandwich concept.

Using ANSI Standards

The need for a standard set of symbols and a standard for their usage in system and program flowcharting has long been recognized by the industry. The ANSI Standards [2] were developed specifically to fill that need. The ANSI symbols, definitions and usages are applicable to the system flowcharting technique presented here with two exceptions: the preparation symbol and the decision symbol. While essential for program flowcharting, these symbols are not used in system flowcharting.

It is important to note that the adoption of ANSI standards does not prohibit the use of symbols not described within that standard. The use of other specialized symbols may be extremely effective in enhancing the communicative power of the flowchart.

THE CONTROL FRAMEWORK

Numerous control frameworks have been suggested for EDP, MIS and DSS systems [5, 8, 9, 11 and 12]. Each is structured around a particular function or perspective, e.g., administration, auditing, designing, operations, organizational structure, or function.

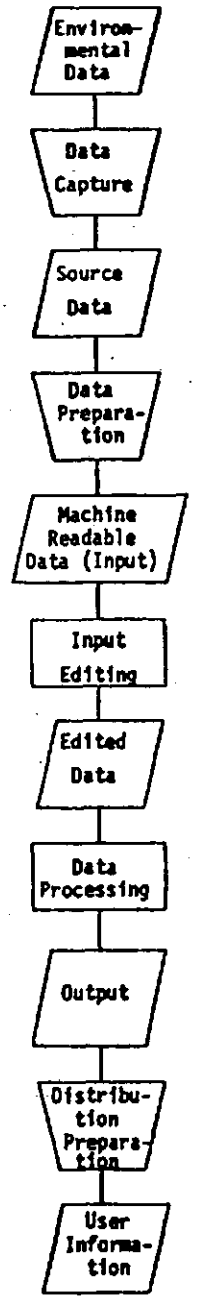


Figure 3: The Basic Data Processing Cycle

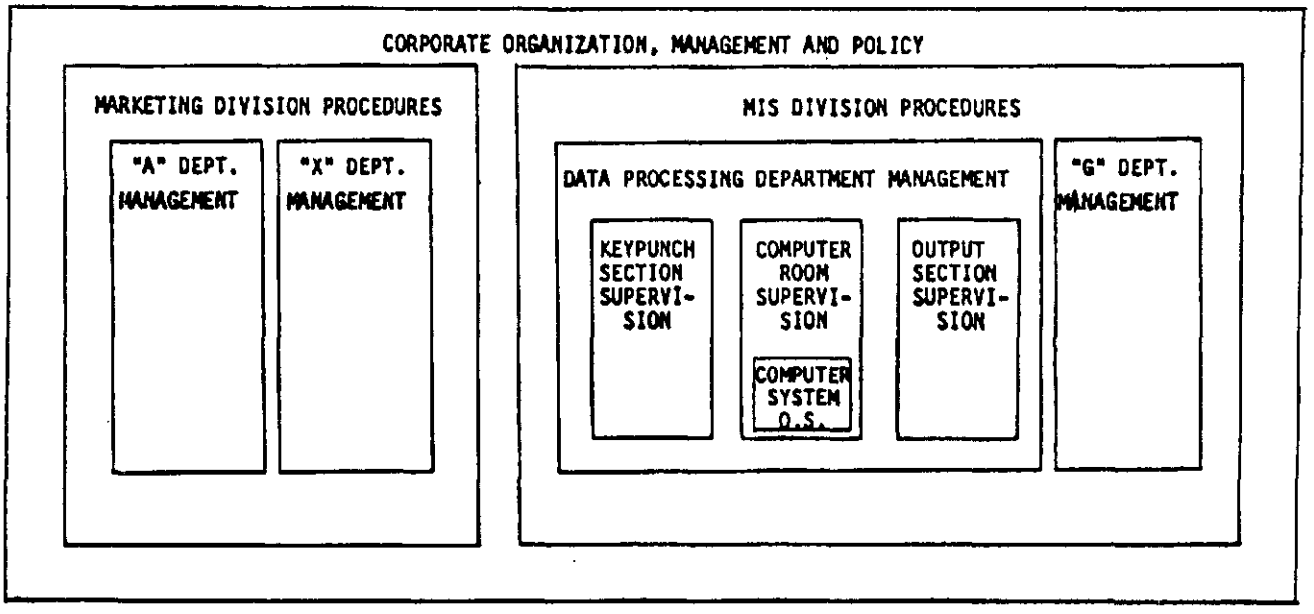


Figure 4: The Organizational Control Structure

Our design definition orientation leads to the selection of a control framework similar to that suggested by Gordon B. Davis [6]. It is structured around two major system aspects, the procedural and processing controls applied to the data flowing within the basic data processing cycle, and the organizational controls applied to the application system.

The Flow of Data

The flow of data through the data processing cycle is an important focus for the system professional and user designer (see Figure 3.) Each of the processes (manual or computer) depicted here provides a vehicle for implementing controls. The controls are specified within the procedures in a manual process and within the programs in a computerized process. The data representations may also function as control vehicles.

Organizational Controls

The system professional and user-designer must function within an organizational structure which imposes many control procedures. Figure 4 illustrates a typical organizational control structure. This structure, reflecting the organization, has a hierarchical character. At the lowest level, any computer data processing must be done within a computer system. The hardware and software which comprise that system have control elements, e.g., the operating system. Data will flow through several sections within the Data Processing Department. Typically each of these sections has its own control procedures and standards. These will operate within the controls for the data processing function established by the data processing department management.

In a similar fashion the systems designer must work within controls established by various departments, through which the flow of data will occur. These departmental controls operate within the framework established by division procedures, which in turn operate

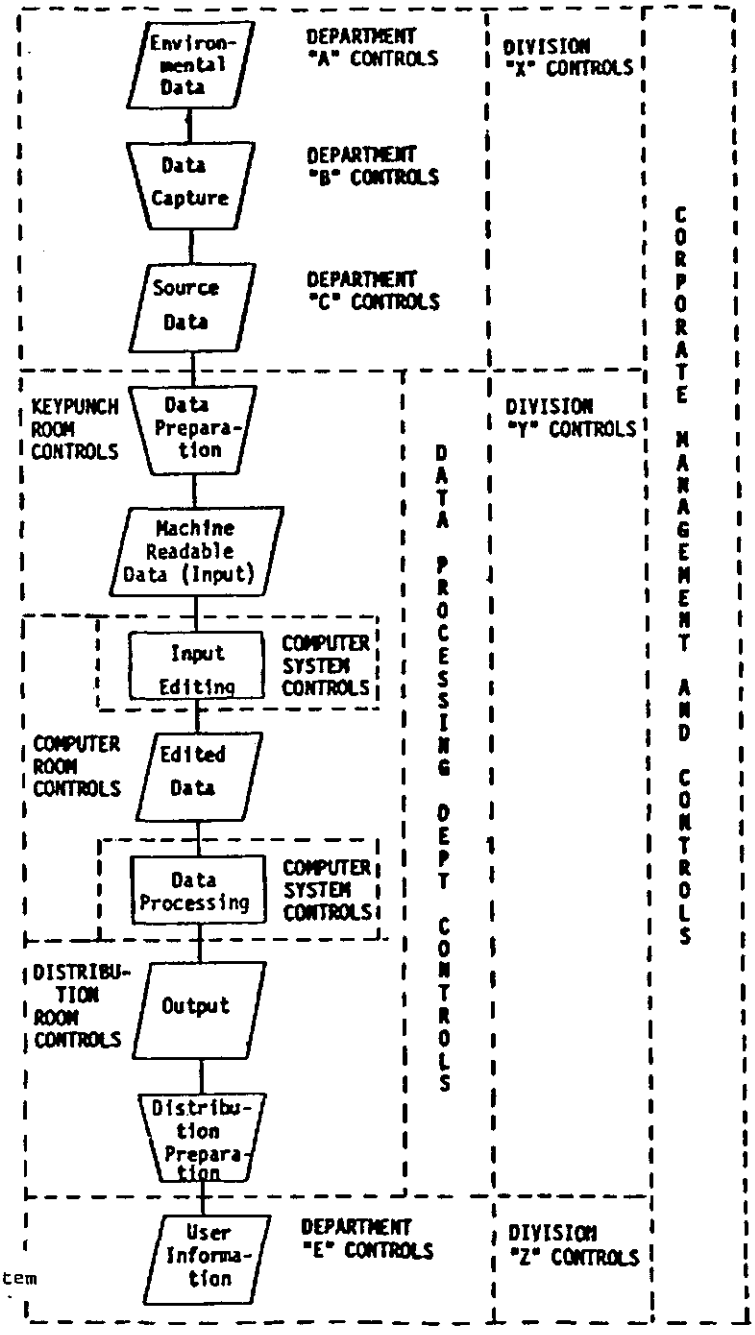


Figure 5: The Control Framework for a Data Processing System

within the corporate policy. Since most application systems cross department and division boundaries, the designer must be cognizant of the various organizational controls which influence the system design.

Integrated Control Model

The integration of the data flow and the organizational controls is depicted in Figure 5. This control framework provides the designer with one framework which is valid through all the phases of the system development life cycle -- from information analysis to post audit. It illustrates which control environments are applicable to each process in the data processing flow. Consequently it is an essential perspective during the design phase, when the designer must specify what controls are to be used and where they are to be applied in the system. This framework underlies the flowcharting techniques described in this paper.

CONTROL FLOWCHARTING

The control flowcharting technique described in this section follows Benjamin's [3] suggestion "that two flow diagrams that present the same process for a system be developed during the design phase -- one for the data flow and one for the control flow." It is predicated on the existence of a system flowchart. While this is a reasonable assumption, it is important to recognize that flowcharts are developed in an iterative fashion. The desired controls may necessitate changes in the original system flowchart. This section provides a description of how a typical control flowchart is developed.

The system flowchart from which the control flowchart will be developed is shown in Figure 6 (which illustrates the same application shown in Figure 1). It has been developed using the technique described in

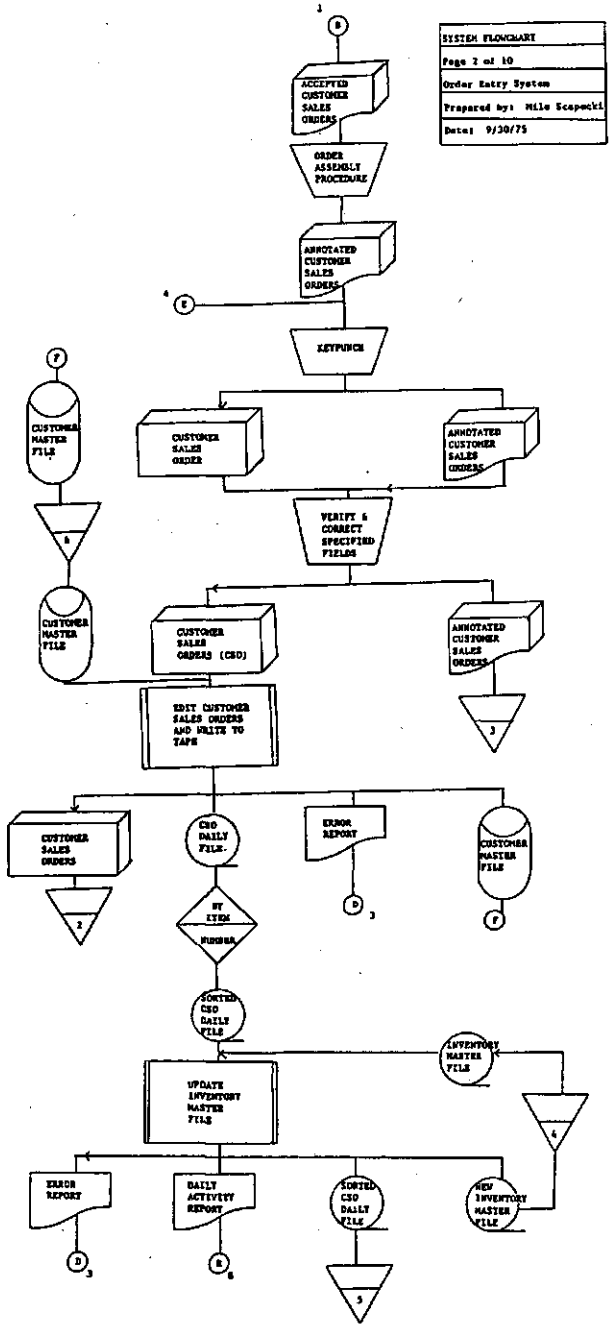


Figure 6

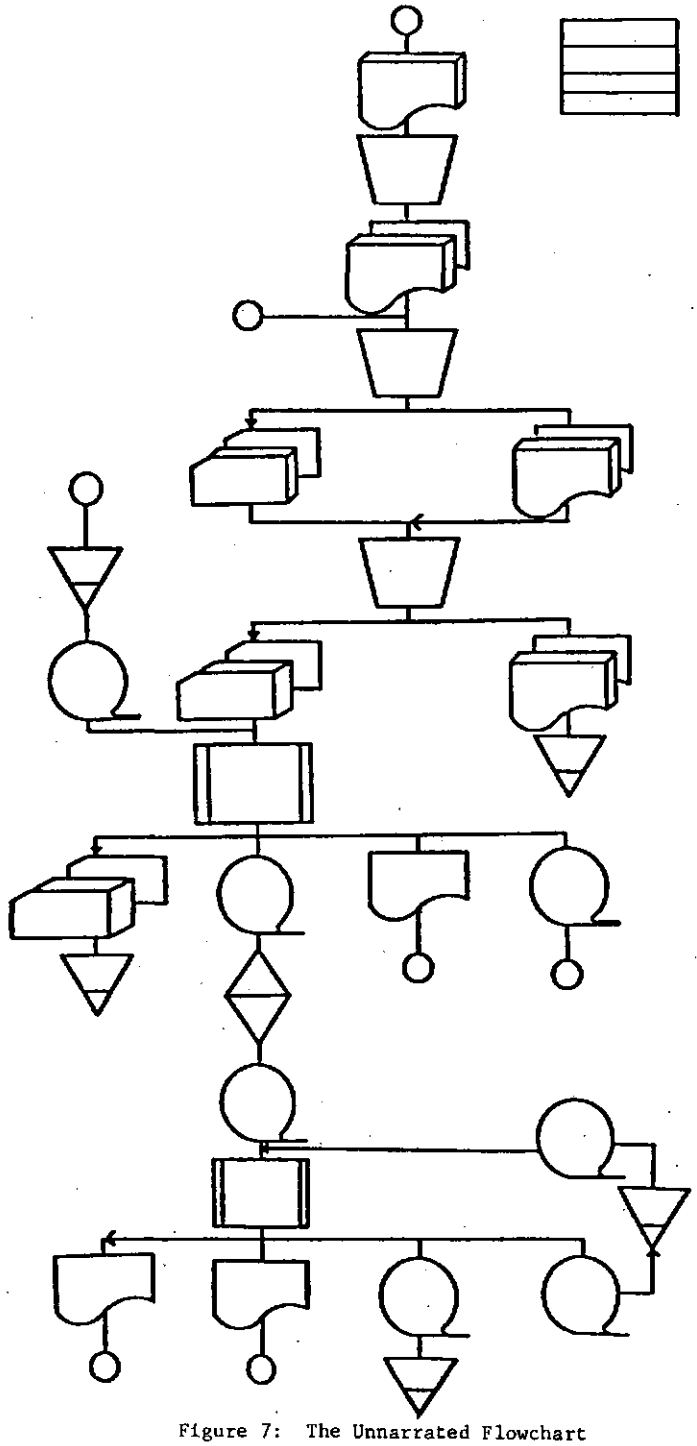


Figure 7: The Unnarrated Flowchart

section two. While constructing the flowchart, after drawing the symbols and flowlines but before narrating the symbols, a photocopy of the flowchart should be made (see Figure 7). This vacant chart provides the physical starting point for the control flowchart, Figure 8. The guidelines that follow implicitly refer to Figures 6 and 8.

Locations for Controls

Working with the system flowchart as a reference, the analyst can begin to specify on the unnarrated copy what controls are appropriate to the system and where they should be located. What set of controls is chosen depends upon the control framework of the organization as discussed in section three, the equipment which will be used, and the particular system being designed. It is beyond the scope of this paper to prescribe the appropriate set of controls for a given system. We provide guidelines for the construction of a control flowchart.

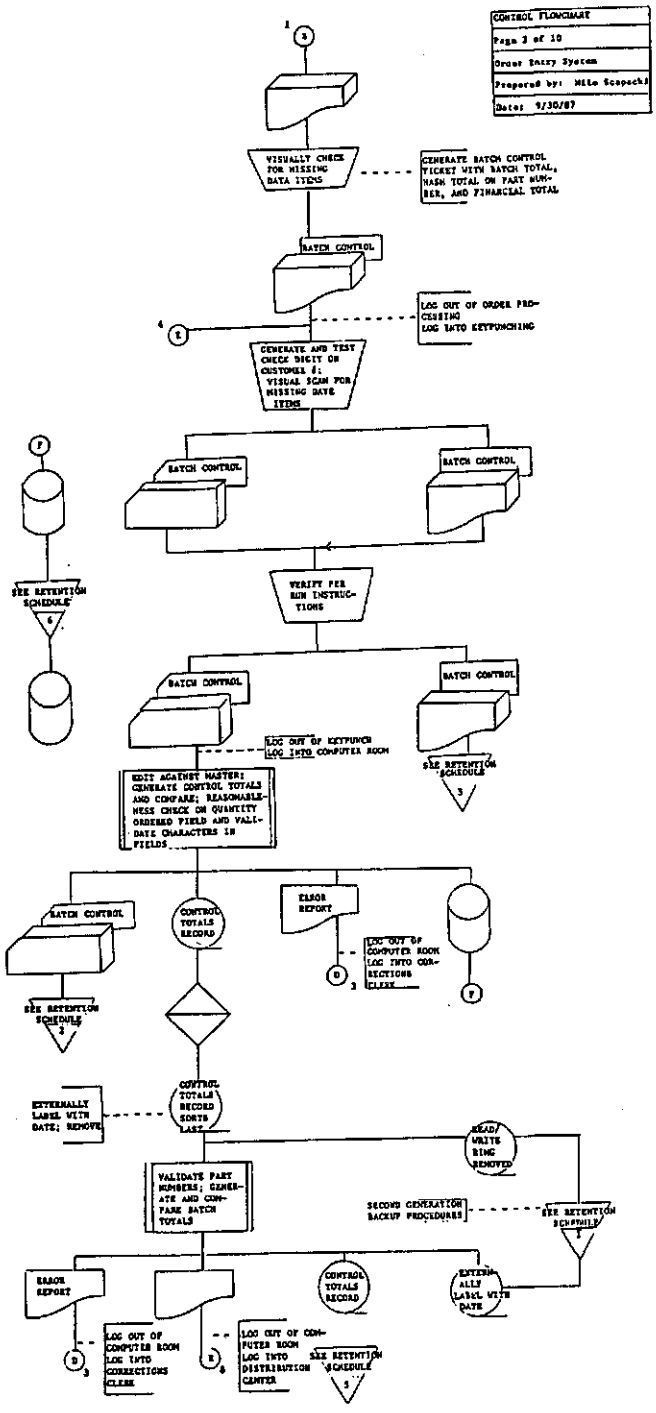


Figure 8

Process Symbols

There are three major types of locations where controls can be supplied. First, a process symbol should be used to specify controls created or exercised during that process. For example, a batch control ticket is created during the order assembly procedure to identify the batch of transactions and also contain control data for later use by both the edit and update procedures. The operation but not the control data itself is described on the unnarrated flowchart in the symbol corresponding to the order assembly procedure of the systems flowchart. The annotation symbol is a convenient documentation aid when the description is too long to easily fit into the process (or medium) symbol.

If no controls are exercised over a process, as in the case of the sort by item number of the CSO daily file, that process symbol remains vacant on the control flowchart. This absence of controls, highlighted by dual charts, is obscured when using only one chart as in Figure 1.

There may be come redundancy between the dual charts. For example, verifying, which is considered both a significant manual (system) process and a control process, appears in both charts.

Control data should not be attached to a process symbol. The operations performed on the control data are described in the process symbol while the control data itself is appended to a medium symbol.

Media Symbols

The second type of location, a data or medium symbol, is used to specify control fields or records that accompany the data and any physical controls applied to the physical medium. To illustrate, no special control fields are implemented on the annotated customer sales orders themselves. As with a process symbol, a blank symbol indicates the absence of controls. However, a control record (the batch control ticket) is added and requires some minor redrawing of the unnarrated chart.

The control flowchart describes the flow and form of control data. The batch control ticket appears in three forms, paper, card, and tape record, and is retained in offline storage. Each change in form requires the intervening process to perform the transformation.

Media are passive, and therefore operations performed on the control data should not appear within the media symbols. It is improper to show a check digit test performed by a floppy disk. Physical controls of the medium itself, not the data it constrains, should appear on the media symbol. A common example is the removal of the write protect ring on a tape.

Flow Lines

The third type of location where controls may be specified is on a flowline. The annotation symbol is used to specify procedural controls associated with the flow of data. The customer sales order deck, after verifying, is input to the edit process. However, it must be physically transported from the keypunch room to the computer room. A movement or change in responsibility may require some control. In this case it is a logging procedure.

A procedural control should be attached to the flowline leaving, not entering, a

medium. This assures the proper association of that procedural control with the medium not the preceding process. This convention is consistent with controlling the flow of data across organizational boundaries as discussed in section three.

Advantages

Keith R. London, in his text, "Documentation Standards," [10] presents the purpose of documentation in the context of four categories: 1) Inter-task/phase communication, 2) Quality control and project control, 3) Historical reference, and 4) Instruction reference. This context is used to structure the following discussion of the advantages of dual flowcharting.

The first category involves facilitating communication within the project team throughout all phases of the systems development life cycle. Given that most project teams have an interdisciplinary composition, some team members will not be familiar with systems techniques and concepts. Consequently, the use of both system and control flowcharts, which simplify the "picture" of flows and of controls, enhances communication. By factoring the control activities from system flow diagram either element may be examined unencumbered by the detail of the other. However, if both elements are of simultaneous interest, the examination and integration of both flowcharts is greatly enhanced by the symbology and flow common to both elements.

London's second category refers to two dimensions of control - control over the quality of the system being designed and control over the process of designing that system. Dual flowcharting aids both of these dimensions.

First, the dual charting technique provides a document exclusively devoted to the controls exercised within the system. Its function is to answer the what, where, and how questions concerning the system's controls. Further, the control flowchart clearly illustrates where controls are not applied in the system. The control flowchart further facilitates quality control by providing a vehicle for communication with interested parties outside of the project team. For example, the control flowchart can be used as a signoff document for internal auditors, the operations manager, and others within the firm with a responsibility for reviewing and approving controls. In the case of user-developed systems the document greatly facilitates communications between the user and system professionals.

The control flowchart enhances project control by enhancing the effectiveness of the documentation package. With better project documentation performance criteria can be better determined at each step in the development cycle. Since performance criteria are the basis of controls, project control is thus facilitated by the use of dual flowcharts.

The third category, historical reference, is used both for modifying existing systems and for developing similar systems. Again, the better the system documentation communicates the essential characteristics of the system the more valuable it becomes in both of these functions. Given the dynamic environment within which systems function, and the iterative nature of systems development, a clear and concise representation of both the flow of data and the application of controls is essential. The development of these charts can greatly reduce the effort required in designing new systems by providing a clear picture of acceptable levels of controls at various stages in the flow and in various departments within the firm. Finally because of the greater simplicity of both the system and control flowcharts, the task of modifying this documentation is greatly reduced.

The final category used to discuss the purposes of documentation is instructional reference. The dual charting technique is especially valuable when serving a general instruction function of communication between the systems specialists and the nonspecialists - particularly the user. Improved user understanding of the flows and controls within the system not only supports better user relations but goes far in enabling the user to use the system intelligently.

Another instructional function enhanced by dual flowcharting is training within the systems group. Easily understood and correct examples of good system design are extremely valuable in training new employees. These flowcharts facilitate rapid understanding of control practices within the systems area and, of equal importance, illustrate the control required in many other functional areas within the organization. Finally, they clearly illustrate to the novice system builder the value and need for good systems documentation.

CONCLUSION

Both the system flowchart and the control flowchart provide important techniques for the systems professional and user-designer in preparing and documenting systems work. Carefully prepared, these flowcharts will enhance the rigor with which the designer can think through the system. This, in turn, typically reduces the errors and increases efficiency within the system. The major contribution of the dual charting technique, however, is its ability to improve the communication of essential aspects of a data-driven application system between interested parties both within and outside of the systems area.

This technique has been used for several years in an educational setting and has proven very valuable to both students and teachers. Little evidence exists on its value in a business data processing environment. However, with the increasing emphasis on good systems documentation and the expanding use of this documentation to communicate with people outside of the systems department, it appears that the advantages offered by the technique are powerful arguments for its use.

BIBLIOGRAPHY

1. Allen, George R., et al., editors, Business Systems, Cleveland, Ohio: Association for Systems Management, 1970.

2. ANSI, American National Standards Institute, American National Standards Flowchart Symbols and Their Usage In Information Processing, New York: 1971, (Document No. X3.5-1970), Sponsored by the Business Equipment Manufacturers Association.

3. Benjamin, Robert I., Control of the Information System Development Cycle, New York: WileyInterscience, a Division of John Wiley and Sons, Inc., 1971, pages 52 and 53.

4. Chapin, Ned, "Flowcharting With ANSI Standard: A Tutorial," Computing Surveys (2:2), June 1970, pages 119-146.

5. Condon, Robert J., Data Processing Systems Analysis and Design, Reston, Virginia: Reston Publishing Company, 1973.

6. Davis, Gordon B., Computer Data Processing, Second Edition, New York: McGraw-Hill Book Company, 1973.

7. Haga, Clifford I., "Procedures Manuals," Ideas for Management, Systems and Procedures Association, Cleveland, Ohio, 1968, pages 127-154.

8. Jancura, Elise G., Audit and Control of Computer Systems, New York: Petrocelli/Charter, 1974.

9. Jancura, Elise G. and Arnold H. Berger, editors, Computers: Auditing and Control, New York: Auerbach Publishers, Inc., 1973.

10. Keith R. London, Documentation Standards, Revised Edition, Petrocelli Books, New York, 1974.

11. Sharratt, John Richard, Data Control Guidelines, Manchester, England: NCC Publications, 1974.

12. Skinner, R.M. and R.J. Anderson, Analytical Auditing, Toronto: Sir Isaac Pitman (Canada), Ltd., 1966.

P. Kolbezen
S. Mavrič
B. Mihovilović
Institut »Jožef Stefan«

UDK 681.3.019

POVZETEK. Večje računalniške zmogljivosti, ki jih zahtevajo današnje aplikacije na področju razpoznavanja govora, procesiranja slik, umetne inteligence in še vrsta drugih, je mogoče iskati le v novih paralelnih sistemih, pri katerih je znatno večja učinkovitost možna le s posebno organizacijo materialne opreme, predvsem v VLSI procesorski tehnologiji, in takšni opremi prilagojeno programsko opremo.

Proizvajalci računalniških sistemov, ki načrtujejo zmogljive vektorske operacije, so izdelali posebne FORTRAN - prevajalnike. Ti razpoznavajo, če se zanka DO lahko zamenja z eno ali več vektorskimi instrukcijami. Zanka DO je tako skalarna predstavitev množice vektorskih operacij in zato more biti izvedena mnogo bolj učinkovito s strojnimi instrukcijami, ki so v takšne namene posebej načrtovane in se zato njihove operacije učinkoviteje izvajajo. Arhitekturne odlike se kažejo z uporabo bodisi cevanih aritmetičnih enot, kot jih srečujemo pri cevanih vektorskih računalnikih (npr. pri CYBER 205 ali CRAY-1), bodisi z večkratnimi procesnimi elementi, kot jih srečujemo pri vektorskem aritmetičnem multiprocesorju (VAMP) ali pri matričnih procesorjih (npr. pri ICL DAP, BSP ali TRANSPUTER ARRAY).

Paralelne vektorske organizacije in podrobnejši opis strukture računalnika ICL DAP so predmet obravnave prvega dela prispevka. Prikazan je tudi stil programiranja, ki ga narekuje takšen računalnik.

V članku je prikazano, kako so lastnosti paralelnih implementacij sekvenčnih kodov na materialni opremi paralelnega sistema pomembne, da prirojeni paralelizem v problemu, ki ga rešujemo, ustreza uporabljeni arhitekturi oziroma organizaciji računalniške materialne opreme.

ABSTRACT. **PARALLEL ARRAY PROCESSORS AND ITS APPLICATION, I.** With new applications such as robotics, speech recognition, artificial intelligence, image processing, etc., the need for faster processing devices becomes more important. The only solution is to employ the new parallel systems. More efficient systems can be attained in first of all with the new VLSI technology and with new organizations of computer hardware and to these ones appropriate software.

All manufacturer of computers designed for efficient operations on vectors of numbers have produced FORTRAN compilers that recognise when a DO loop can be replaced by one or several vector instructions. This is the recognition by software that a particular DO loop is just the scalar representation of a set of vector operation, and can therefore be executed much more effectively by machine instructions that are especially engineered to perform such operations efficiently. The architectural features used are then either the pipelined arithmetic units in the case of pipelined vector computers (e.g. CYBER 205 and CRAY-1) or the replicated processing elements in the case of vector arithmetic multiprocesor (VAMP) or processor array (e.g. ICL DAP, BSP and TRANSPUTER ARRAY).

In the present first part of this work the basic categories of parallel computers will be described and the detailed structure of ICL DAP will be given in more detail, together with an overview of the programming style that is requires. In the second part the optimization algorithm based on matrix computations as a tool for implementations of application algorithms will be continued.

In the paper is shown that the behaviour of parallel implementations of sequential codes on parllel hardware depends critically on a careful match between the innate parallelism of the problems, the algorithm and the hardware.

1. UVOD

Zadnja leta so tehnološke spremembe in številne inovacije dramatično zmanjšale ceno računalniških zmogljivosti. Na vsakih nekaj let so se pojavili bistveno hitrejši in cenejši procesorji. Posebne prednosti teh procesorjev so se pričele kazati pri sestavljanju le-teh v različne paralelne sisteme. Sistemski programerji so dobili spodbudo, da so modificirali obstoječo sistemsko programsko opremo enoprocesorskih sistemov tako, da je z njo mogoče kar najbolje izkoriščati prednosti novih konceptov paralelnih sistemov.

Poskušali bomo kolikor mogoče jasno predstaviti rezultate na novejšem področju računalniških arhitektur oziroma organizacij, ki naj bi pokazali bistvene prednosti rabe paralelnih procesorskih sistemov. Prednosti se kažejo v novih aplikacijah, kot so robotika, razpoznavanje govora, umetna inteligenca, procesiranje slik itd., kjer so postale zahteve po hitrejših procesorskih napravah vse pomembnejše. Zaradi zaporednega izvajanja instrukcij, ki se izvaja-

jo druga za drugo, je razvoj sekvenčnih arhitektur skoraj obstal. Rešitve je nadalje mogoče iskati le v novih paralelnih sistemih, pri katerih je znatno večja učinkovitost možna le s posebno organizacijo materialne in njej ustrezne programske opreme.

Polpretekla zgodovina kaže, da je paralelizem mogoče vpeljati na različnih nivojih, ki jih lahko opredelimo takole:

1. Nivo posla
   1.1., med posli;
   1.2., med fazami posla;
2. Programski nivo
   2.1., med deli programa;
   2.2., znotraj DO zank;
3. Instrukcijski nivo
   3.1., med fazami izvajanja instrukcije;
4. Aritmetični in bitni nivo
   4.1., med elementi vektorskih operacij;
   4.2., v sklopu aritmetičnih logičnih vezij;

Proizvajalci računalniških sistemov, ki načrtujejo učinkovite vektorske operacije, so izdelali posebne FORTRAN - prevajalnike. Ti razpoznavajo, če se zanka DO lahko zamenja z eno ali več vektorskimi instrukcijami. Zanka DO je tako skalarna predstavitev množice vektorskih operacij in zato more biti izvedena mnogo bolj učinkovito s strojnimi instrukcijami, ki so v takšne namene posebej načrtovane in se zato njihove operacije kar najučinkoviteje tudi izvajajo. Arhitekturne odlike se kažejo z uporabo bodisi cevanih aritmetičnih enot, kot jih srečujemo pri cevanih vektorskih računalnikih (npr. pri CYBER 205 ali CRAY-1), bodisi z večkratnimi procesnimi elementi, kot jih srečujemo pri vektorskem aritmetičnem multiprocesorju (VAMP) ali pri matričnih procesorjih (npr. pri ICL DAP ali BSP).

Paralelne vektorske organizacije bodo predmet prvega dela naše obravnave, medtem ko bo drugi del prispevka posvečen uporabi sistema s takšno organizacijo. Med drugim bomo obravnavali optimizacijski postopek, ki je zasnovan na matričnem računu in se lahko uporablja kot učinkovito orodje za implementacijo danega algoritma na paralelnem računalniku.

Videli bomo, da so lastnosti paralelnih implementacij sekvenčnih kodov na materialni opremi paralelnega sistema močno odvisne od prilagoditve prirojenega paralelizma v problemu, ki ga rešujemo, uporabljeni računalniški materialni opremi. To odvisnost lahko v določeni meri ugotavljamo na primeru, pri katerem uporabljamo paralelni računalnik za reševanje problemov, ki so izrazito sekvenčni. Drug podoben primer je, če na specifičnem paralelnem procesorskem sistemu izvajamo takšen algoritem, v katerem sicer obstaja nek paralelizem, je pa ta povsem drugačne narave, kot paralelizem algoritmov, za izvajanje katerih je bil specifični paralelni sistem tudi zgrajen. Zato bomo v prispevku sledili ugotovitvam učinkovitosti rabe posameznih arhitektur pri reševanju problemov določenega razreda.

2. PARALELNI SISTEMI

Klasifikacija procesorskih sistemov je dandanes ponovno vse manj zadovoljivo rešena. Znana avtorja na tem področju sta Flynn (1972) in Shore (1973). Problem ustrezne klasifikacije tiči v tem, da več dobro vpeljanih sodobnejših arhitektur, predvsem učinkovitih cevanih računalnikov, ne sodi dovolj jasno niti v en, niti v drug razred poznanih klasifikacij, oziroma, da tako kot računalnik ICL DAP, le-ta enako dobro ustreza hkrati dvema razredoma. Alternativni pristop do primerne klasifikacije je, da določimo razrede glede na osnovne načine, na kakršne se paralelizem javlja v arhitekturi nekega računalnika. Tako lahkogovorimo o paralelizmu cevanja, kopiranja oziroma vektorske porazdeljenosti procesorjev, funkcionalnosti ali o multiprocesorskem paralelizmu:

(1) **Paralelizem cevanja (pipelining)** je zasnovan na tehniki t.im. zbirne linije, ki povečuje zmogljivost aritmetične ali krmilne enote;
(2) **Paralelizem funkcionalnosti** je zasnovan na več med seboj neodvisnih enot, ki lahko izvajajo različne funkcije (kot so logične funkcije, funkcije seštevanja in množenja) in sočasno izvršujejo operacije nad različnimi podatki.
(3) **Vektorski paralelizem** je zasnovan na polju identičnih procesnih elementov pod skupnim nadzorom ene same krmilne enote. Ti elementi sočasno izvajajo enake operacije nad različnimi podatki, ki so shranjeni v njihovih zasebnih pomnilnikih. To so t.im. koračne (lock-step) operacije z zaklepanjem;
(4) **Multiprocesorski paralelizem** izkorišča večje število procesorjev, od katerih izvaja vsak svoje instrukcije, medtem ko navadno med seboj komunicirajo preko skupnega pomnilnika.

Flynn /1/ je razdelil paralelne računalnike, ki procesirajo več podatkovnih tokov hkrati v dve kategoriji. To so večinstrukcijski večpodatkovni stroji MIMD (Multipl-Instruction Multiple-Data) in enoinstrukcijski večpodatkovni stroji SIMD (Single-Instruction Multiple-Data).

Vsaka od obeh kategorij ima svoje značilnosti. Oglejmo si jih na kratko.

Računalnik MIMD ima navadno majhno število nepretirano učinkovitih in med seboj povezanih procesorjev, ki lahko sodelujejo med seboj ali pa delujejo neodvisno drug od drugega. Zato je potrebno, da so neodvisna programska opravila kodirana in dodeljevana posameznim procesorjem. Posebna skrb mora biti posvečena sinhronizaciji dodeljevanja informacij med procesorji, ko eden od procesorjev zahteva informacijo od drugega procesorja. Dokaj jasno je, da pri obsežnejših opravilih in pogostejši sinhronizaciji nekateri procesorji čakajo brez dela, dokler drugi procesorji ne dokončajo svojega opravila. Primer stroja takšnega tipa je Loughborough-ov računalnik Neptun, ki je zasnovan na štirih miniračunalnikih Texas Instrument 990/10. Ta računalnik in njegovo uporabo je podrobneje opisal R.H.Barlow /1 /.

Računalniki, vodeni s tokom podatkov, ki pripadajo kategoriji strojev MIMD, predstavljajo ločeno podkatergorijo. Vzrok ločitve tiči v drugačnem postopku vodenja, ki temelji na drugih in ne na klasičnem von Neumannovem konceptu. Pri klasičnem tj. sekvenčnem računalniku predstavlja program, ki se na njem izvaja, neko zaporedje (sekvenco) instrukcij. V računalniku MIMD pa se več sekcij kode lahko izvaja paralelno na različnih procesorjih. V vsaki od sekcij lahko še nadalje obstaja vzporedje, ki pa je nevidno programerju, ker se le-to pojavlja znotraj posameznih instrukcij.

Pristop vodenja s tokom podatkov se naslanja na direktno predstavitev z grafom. Vzemimo enačbo:

$$y = a.x^3 + b.x^2 + c.x + d$$

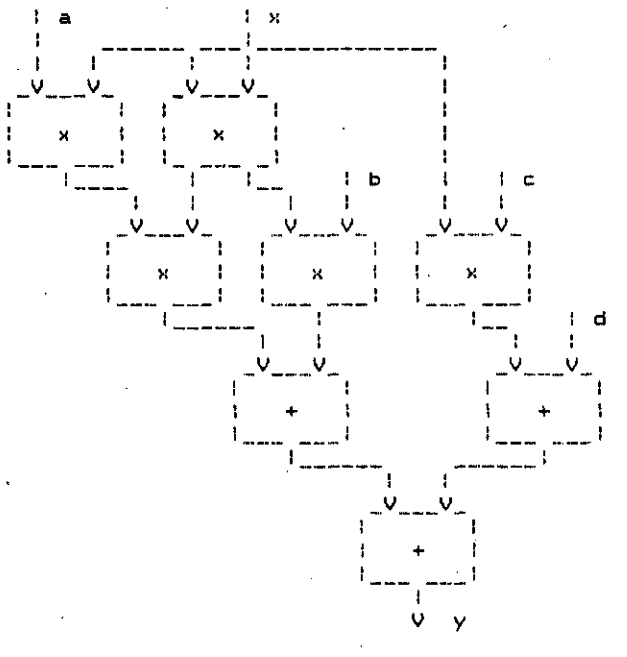
Gornjo enačbo moremo izraziti na običajen način v visokonivojskem programu. Moremo pa je predstaviti tudi direktno z grafom, ki ga vidimo na sliki 1. V grafu na tej sliki predstavlja vsako vozlišče eno od osnovnih operacij (tj. množenje ".", ki je na sliki označeno z "x", in sešte-

vanje, ki je označeno z "+").

Vidimo, da so vozlišča grafa povezana s pušči-cami, ki so obrnjene v smeri toka informacij: vhod-izhod. Informacija potuje od izhoda vozlišča, ki je pravkar izvršilo svojo funkcijo (tj. določeno operacijo), k vhodu vozlišča, ki bo svojo funkcijo šele pričelo izvajati. Serijsko - paralelna odvisnost med operacijami postane očitnejša pri predpostavki, da lahko dano vozlišče izvrši svojo funkcijo samo tedaj, ko so prisotni vsi njeni vhodi. Operacije (funkcije) so v tem primeru vodene le s tokom podatkov. Takšen pristop izvajanja paralelizma ima prednost v tem, da programer uporablja pri implementaciji stroja, ki je voden s tokom podatkov, dokaj običajen jezik. Podrobnosti o takšnem stroju najdemo na primer v delih Johnsona /3/ in Sauberja /4/ iz firme Texas Instruments in v delih da-Silva in Woodsa /5/, ki sta delala na Manchester računalniku, pa tudi v seriji člankov na temo podatkovno vodenih arhitektur v zadnjih dveh letih v tem časopisu (Informatica, letniki 85,86 in 87).

V popolnem nasprotju s stroji MIMD imajo stroji SIMD navadno veliko število (včasih tudi več tisoč) manj učinkovitih procesorjev, ki so prav tako, vendar bolj strogo povezani med seboj. Očitnejša razlika teh strojev v primerjavi s stroji MIMD je, da so njih procesorji praviloma sposobni opravljati le enaka osnovna opravila. V nadaljevanju bomo uporabljali izraz **vektorski procesor** izključno za računalnike razreda SIMD, ki uporabljajo klasičen (neasociativen) pomnilnik z naključnim dostopom (RAM), in izraz **asociativni procesor** za računalnike iz razreda SIMD, ki uporabljajo asociativen pomnilnik. Med vektorske procesorje sodijo navadno oboji: vektorsko cevani in matrični računalniki. Tipična primera takšnih računalnikov sta računalnika GRAY1 in ICL DAP. Prvega uvrščamo med vektorsko cevane, drugega pa med matrične računalnike.

V pojasnilo naj posebej omenimo, da Gray-1 nima veliko število procesorjev. Kljub temu ga uvrščamo med stroje SIMD. Paralelni operandi so

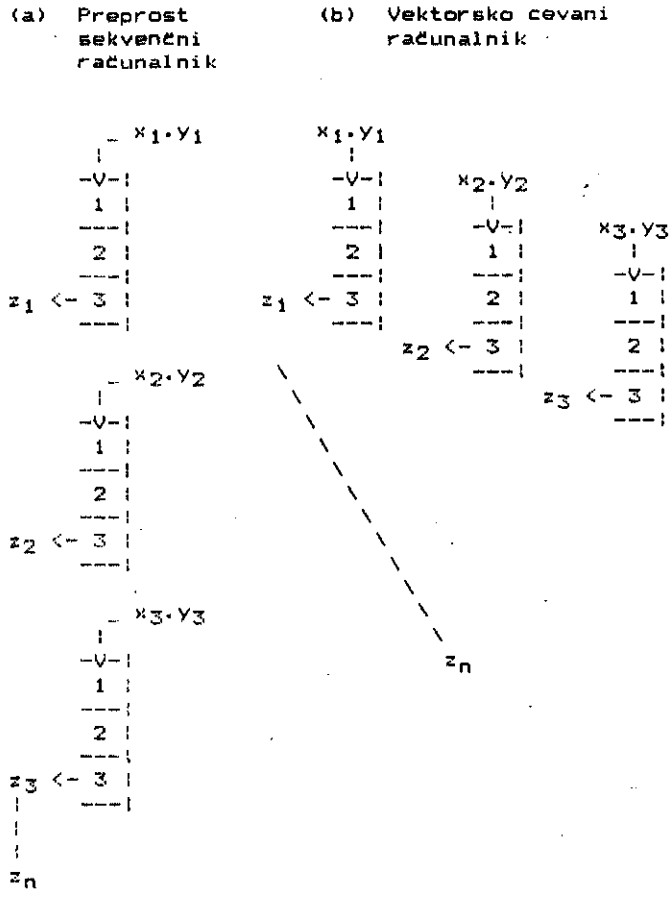


Slika 1. Krmiljenje s tokom podatkov

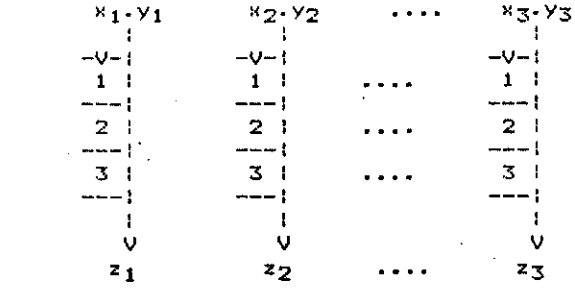
namreč omejeni v takšnem smislu, da opravljajo enake operacije. Kot primer vektorskega cevanja vzemimo element računanja pomične vejice, tj. element množenja dveh vektorjev x in y z rezultatom z, pri čemer imajo vsi vektorji dolžino 64. Rezultat dobimo z izvajanjem naslednjih korakov:

(1) Izvršimo produkt posameznih komponent
(2) Dodamo eksponente
(3) Normaliziramo in zaokrožimo rezultate

Pri preprostem sekvenčnem računalniku se vse tri stopnje izvajajo zaporedno za vsak par operandov (vektorskih komponent). Pri tem se moramo zavedati, da med izvajanjem seštevanja eksponentov v 2.koraku, stopnja 1 izvajanja algoritma miruje; podobno, ko se rezultati v 3.koraku normalizirajo, mirujeta stopnja 1 in 2. V vektorsko cevanem računalniku, kot je npr. Gray-1, pa obstaja več segmentiranih funkcionalnih enot, ki lahko delujejo paralelno. Na



(c) Paralelni vektorski procesor

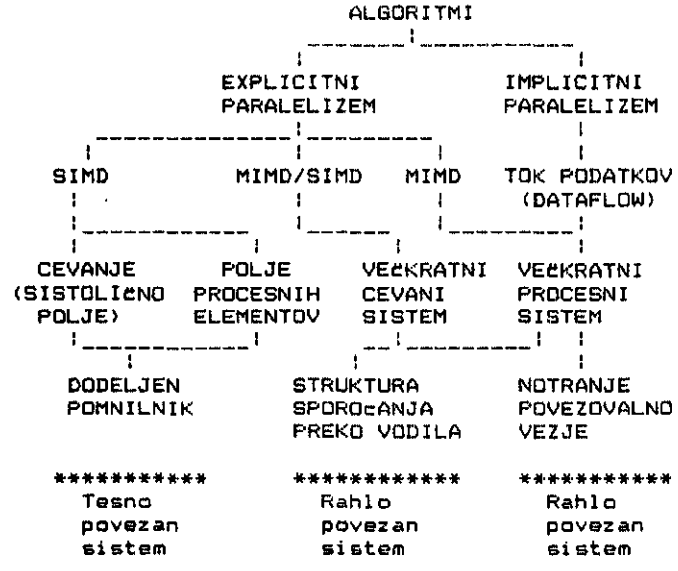


Slika 2. Cevanje pri vektorsko cevanem in paralelnem vektorskem procesorju

primer, ko se izvrši množenje segmentov x_1 in y_1, se prične seštevanje eksponentov segmentov x_1 in y_1, toda istočasno starta tudi množenje naslednjih dveh segmentov x_2 in y_2, itd. Mimogrede povedano, tudi srednji in zelo hitri sekvenčni stroji izkoriščajo prednosti cevanja. Obravnavani primer je prikazan na sliki 2. (za n=64).

Pri sekvenčnem stroju so potrebni trije urini impulzi za izračun vsakega "delnega" rezultata, kar pomeni 192 impulzov, da se izračuna celoten vektor z. Drugače je pri vektorskem cevanju, kjer je po prvih dveh impulzih potreben le še en sam urin impulz na "delni" rezultat. To pomeni skupaj 66 impulzov, da se izračuna celoten vektor z. V primeru matričnega procesiranja pa dobimo vseh n (delnih) rezultatov v treh urinih impulzih. Torej lahko zaključimo: če primerjamo vse tri možne načine množenja n = 64 komponentnih vektorjev in uporabimo:

a) sekvenčen računalnik, potrebujemo 3n=192 impulzov,
b) vektorsko cevani računalnik, potrebujemo 2+n = 66 impuzov,
c) paralelni vektorski procesor, potrebujemo 3 impulze.



Slika 3. Klasifikacija paralelnih računalniških arhitektur.

Podrobnosti o računalniku Gray-1 najdemo med drugimi tudi v delih /6/ in /7/.

Podrobnejšo klasifikacijo strojev je mogoče podati bodisi z vidika organizacije njenih sestavnih delov, v katerih se vzporedje pojavlja, bodisi z vidika uporabljenega načina krmiljenja. Slednja klasifikacija je obravnavana v delu B.Robiča /8/. V nadaljevanju pa se bomo omejili predvsem na prvo omenjeno klasifikacijo, ki je osnovana na paralelnem procesiranju bitov ali/in besed in na številu uporabljenih krmilnih enot. Vse pogosteje pa dandanes srečujemo tudi sisteme, ki vključujejo oba načina vodenja: podatkovnega in nepodatkovnega, od katerih slednji sloni na klasični von-Neummanovi arhitekturi. Takšen "mešani" sistem je npr. wavefront procesor. Upoštevajoč obe omenjeni klasifikaciji, je mogoče klasifikacijo strojev še bolj nadrobno razdelati. Glede na način izvajanja paralelizma v algoritmu, lahko paralelne arhitekture klasificiramo /9/ po shemi na sliki 3.

2.1. Vektorski in asociativni procesorji

Glede na zgoraj omenjene vidike procesiranja pa lahko paralelne procesorske sisteme razdelimo v šest razredov, in sicer:

BPWSAR - bitno paralelni (BP), besedno serijski (WS), vektorski (AR) procesorji:UNGER, SOLOMON, VAMP, CDC 7600, GRAY-1, ILLIAC, BSP in NASF;

WPBSAR - besedno paralelni (WP), bitno serijski (BS), vektorski (AR) procesorji: CLIP, DAP, STARAN in MPP;

WBSPAS - besedno (W) in bitno (B) serijski (S) in paralelni (P) asociativni (AS) procesorji ali tim. ortogonalni procesorji:OMEN 60. Računalniki tega razreda se lahko združujejo v ustrezne MSIMD (tj. večkratne SIMD računalnike): MAP, PM^4;

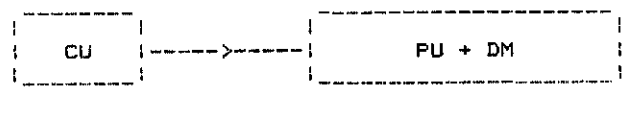
BPWSAS - bitno paralelni (BP), besedno serijski (WS), asociativni (AS) procesorji (zasnova procesorja PEPE);

WPBSAS - besedno paralelni (WP), bitno serijski (BS), asociativni (AS) procesorji: STARAN, RELACS;

UNCNAS - nepovezani (UNCN) asociativni (AS) procesorji z besednimi rezinami (bitno serijski): PEPE;

CNAR - povezani (CN) vektorski (AR) procesorji z besednimi rezinami (bitno serijski): ILLIAC IV. Med pridružene računalnike MSIMD tega razreda lahko prištevamo sistem PHOENIX.

Vsi vektorski procesorji podpirajo koncept ločenih podatkovnih pomnilnikov (DM) in procesnih enot (PU), ki so povezani med seboj z nekim podatkovnim vodilom ali stikalnim elementom. Pri tem je nepomembno, da imajo nekatere implementacije enobitnih strojev WPBSAR procesno enoto in podatkovni pomnilnik realizirana na isti vtični enoti (tiskanini). Takšen procesor je npr. ICL DAP. Nevektorski procesorji pa predstavljajo alternativni pristop k porazdeljeni procesorski logiki, ki je prenešena na pomnilnik. Imenujemo jih "Vsebinsko naslovljivi paralelni procesorji" (CAPP) ali "Asociativni procesorji" (LIMA) in zavzemajo obsežno področje procesorjev od najpreprostejših asociativnih pomnilnikov do zelo kompleksnih asociativnih procesorjev /10,11/. Shemo arhitekture te vrste procesorjev iz razreda LIMA prikazuje slika 4.



Slika 4. Arhitektura procesorja iz razreda LIMA.

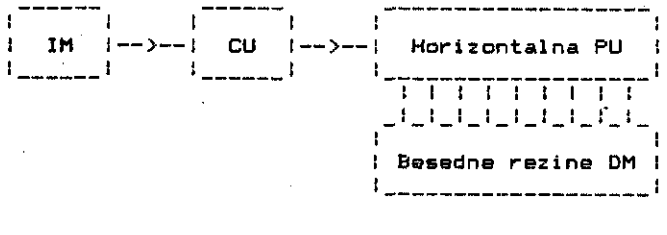
Področje aplikacij strojev iz razreda LIMA, ki se odlikujejo po izredno hitrih operacijah in po lahkem programiranju, je zelo veliko. Med napogostejše aplikacije sodijo: procesiranje radarskih podatkov, testiranje povezanosti, ki jo srečujemo v determinističnih implikacijah semantičnih mrež umetne inteligence, pravopisno popravljanje, digitalno diferencialno analiziranje analognih podatkov (kar predstavlja bistveno bolj učinkovito reševanje problemov, ki jih sicer rešujejo analogni računalniki), nadzorovanje zračnega prometa, urejanje, časovno

vhodno/izhodno dodeljevanje med terminali in glavnim računalnikom in reševanje relaksacijskih problemov v fiziki. Pogosto pa se asociativni paralelni procesorji uporabljajo tudi za opravljanje številnih specifičnih funkcij v konvencionalnih računalnikih.

Vsebinsko naslovljiv paralelni procesor CAPP je zasnovan na vsebinsko naslovljivem pomnilniku CAM, ki mu je dodana sposobnost paralelnega vpisovanja v besede, ki so za ta namen posebej označene. Lahko se spreminja celotna vsebina besed, ali le del besede, ali celo samo posamezni biti v označeni besedi. V takšnem primeru govorimo o sposobnosti pomnilnika z večkratnim zapisom. Po tej sposobnosti se loči CAM od CAPP. Le-ta omogoča izvajanje paralelne aritmetike, sestavljeno iskanje in v splošnem emulacijo vektorskih računalnikov, kot na primer računalnika ILLIAC. Prvi predlog pomnilnika CAM je dal Slade že leta 1956, v letu 1972 pa je Goodyear Aerospace Corporation poslal na tržišče računalnik STARAN, kot prvi komercialno dosegljiv CAPP.

Nadalje si oglejmo še ostale razrede paralelnih procesorjev in vsaj po enega predstavnika vsakega razreda.

BPWSAR. Procesor je konvencionalne von Neumanove arhitekture z eno samo krmilno enoto (CU), procesno enoto (PU), krmilnim ali instrukcijskim pomnilnikom (IM) in podatkovnim pomnilnikom (DM). Enojni DM pri čitanju predaja vse bite neke besede PU, da jih le-ta paralelno procesira. PU lahko sestavlja več funkcionalnih enot, ki so lahko tudi cevane. Zato pripadajo temu razredu tako cevani skalarni računalniki (kot je npr. CDC 7600) in cevani vektorski računalniki (npr. GRAYi), ki se odlikuje po svoji preprosti arhitekturi. Shemo arhitekture računalnikov tega razreda ponazoruje slika 5.



Slika 5. Arhitektura procesorja iz razreda BPWSAR

UNGER (1958) je zasnoval računalnik za reševanje prostorskih problmov, predvsem za aplikacije kot so npr. razpoznavanje vzorcev. Ta računalnik, imenovan tudi "prostorski računalnik", ima dvodimenzionalno polje PE pod skupnim nadzorom. V računalniku **SOLOMON** je bil vpeljan koncept t.im. "zaklepanja po korakih" v operacijah strojev SIMD. Koncept je predlagal Slotnick (1962), realiziran pa je bil šele v seriji strojev ILLIAC in še nekaterih kasnejših strojih tipa SIMD. Tri leta kasneje je bil zgrajen vektorski aritmetični procesor VAMP (Senzing in Smith), ki je sestavljen iz linearnih vektorjev PE z dodeljenimi jim pomnilniškimi moduli in cevano aritmetično enoto. Vsaka PE je virtualni procesor, ki vsebuje samo nekaj delovnih registrov. Cevani vektorski procesor pa je bil načrtovan z namenom, da zmanjša ceno materialne opreme, potrebne za vektorsko procesiranje.

Prvi model računalnika **GRAY-1** je bil zgrajen pri Gray Research Inc. kot najhitrejši računalnik na svetu in dobavljen laboratoriju "Los Alamos Scientific Laboratory" leta 1976. To je tudi prvi komercialno dosegljiv cevani vektorski procesor. Ima 12 funkcionalnih enot, sedaj že vse cevane, hitrjšo uro 12,5 ns, 16 bank bipolarnega pomnilnika z 10^6 besed in 50 ns ciklom ter osem 64-bitnih vektorskih registrov za pomnenje 64-ih števil v pomični vejici. Aritmetične operacije nad temi vektorji izvaja s približno 32 strojnimi ukazi. Tri funkcionalne enote so namenjene vektorskim operacijam (pomiku, logičnim operacijam in seštevanju), tri enote pa skalarnim operacijam (seštevanju v plavajoči vejici, množenju in recipročni aproksimaciji) /7/.

Burroughs Corporation je igral vodilno vlogo pri razvoju vektorskih paralelnih procesorjih. Ta se je pričel s procesorjem ILLIAC IV in nadaljeval s paralelnim procesorjem PEPE. Kmalu mu je sledil BSP s komercialnimi težnjami na perspektivnem tržišču, ki ga je predstavljalo že takrat ekspanzivno splošno znanstveno - raziskovalno okolje.

Projekt **ILLIAC IV** je imel cilj, da se razvije visoko parelelni računalnik z velikim številom aritmetičnih enot, ki bi izvajale vektorske ali matrične izračune s hitrostjo reda 10^9 operacij na sekundo. Da bi se dosegla takšna zmogljivost sistema, je bil računalnik prvotno načrtovan z 256 PE-mi, katere naj bi nadzorovale 4 centralne procesne enote (CU). Zarade previsoke cene in nesprejemljivih zakasnitev operacij dodeljevanja je bil prvotni sistem okrnjen na eno četrtino. Tako končni produkt, ki je namenjen predvsem reševanju parcialnih diferencialnih enačb (v numerični obdelavi vremenoslovnih kart za napovedovanje vremena, v nuklearnih raziskavah in v drugih številnih aplikacijah), sestavlja le polje 8 x 8 po 64-bitnih procesnih elementov za računanje s plavajočo vejico pod nadzorom ene same CU. S tako okrnjenim sistemom so dosegli hitrost 200 milijonov operacij na sekundo. Vsaki PE je dodeljen pomnilnik z 2K besedami, ki dela po načinu koračnega zaklepanja z najbližjimi sosednimi povezavami.

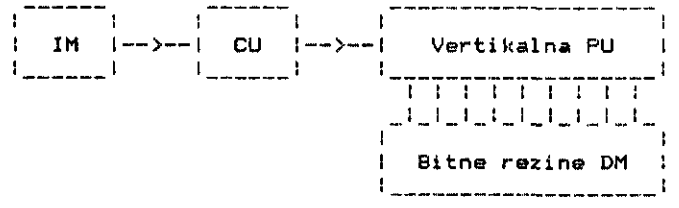
Eden od problemov, ki so nastopali pri uporabi eksperimentalnega prototipa ILLIAC IV, so bile zakasnitve pri prenašanju podatkov na dolge razdalje preko polja PE zaradi omejitev v številu najbližjih sosednih povezav med 64-imi procesorji in njihovimi 64-imi bankami pomnilnikov (PEM). Zato je Burroughs v svojih komercialnih produktih računalnik ILLIAC pričel proizvajati v "okrnjeni" izvedbi z 16-imi procesorji in 17-imi pmnilniškimi bankami pod oznako **BSP** (Burrougs Scientifics Processor). Zaradi manjšega števila procesorjev je bilo mogoče komunikacije izvesti preko posebnega "uvrstitvenega vezja" med poljubnima paroma procesorja in pomnilniške banke. Večje število pomnilniških bank v primerjavi s številom procesorjev dopušča uporabo algoritmov preslikav, ki zmanjšujejo število pomnilniških konfliktov. Le-ti lahko sicer močno narastejo v pogostih manipulacijah z matrikami. PE so procesorji, ki omogočajo računanje s plavajočo vejico, in so serijsko organizirani. Seštevanje ali množenje izvajajo paralelno in dajo 16 rezultatov tovrstnih operacij iz polja PE v 320 nsek. S skrbnim prekrivanjem čitanja, vpisovanja in aritmetike, skupaj s povezovanjem viška PEM s PE, računalnik BSP učinkovito preprečuje ozka grla in dosega pri reševanju večine problemov, ki jih rešuje, maksimalno procesno hitrost 50 Mflop/s.

V ta razred prištevamo tudi novejši sistem **NASF** za potrebe NASA: Numerical Aerodynamic Simulation Facility. Sistem je predlagal Stevens leta 1979. Načrtal ga je CDC na osnovi izpopolnjenih štirikratno cevanih računalnikov CYBER 205, ki delajo po načinu t.im. "koračnega zaklepanja",

z dodatnim petkratnim cevanjem kot pripravljeno rezervo, ki se elektronsko vključi v primeru, da se odkrije napake v delovanju sistema. Vsako cevanje lahko da en 64-bitni rezultat ali dva 32-bitna rezultata na vsakih 8 nsek. V vsakem primeru pa je rezultat dobljen v največ treh operacijah. Zato je pri tem računalniku dosežena doslej največja hitrost računaja v aritmetiki s plavajočo vejico, in sicer 3 Gflop/sek. K temu pripomore tudi hiter skalarni procesor z 16 nsek uro.

WPBSAR. Procesor tega razreda se v bistvu razlikuje od procesorja iz prejšnega razreda samo v tem, da se pri čitanju DM dostavi bitna rezina vseh besed v pomnilniku, namesto da bi se dostavili vsi biti ene besede. Zato je PU organizirana tako, da izvaja vse operacije v bit-serijskem načinu. V primeru, da je pomnilnik dvo-dimenzionalno polje bitov z eno besedo na vrstico, čita računalnik tega razreda vertikalno rezino bitov, medtem ko čita računalnik iz prejšnega razreda horizontalno rezino. Primera strojev iz tega razreda sta ICL DAP in STARAN, shema arhitekture teh procesorjev pa je prikazana na sliki 6.

Medtem, ko si bomo računalnik ICL DAP podrobneje ogledali v naslednem poglavju, bomo na tem mestu nekaj pozornosti posvetili računalnikoma CLIP in MPP.

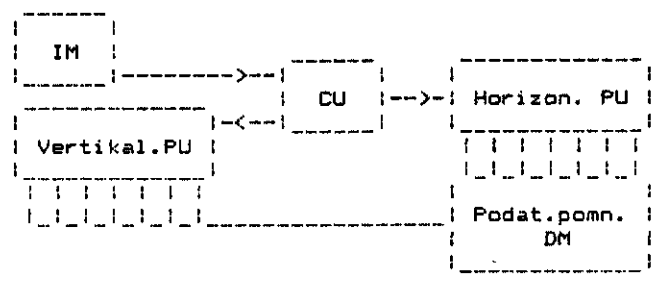


Slika 6. Arhitektura procesorja iz razreda WPBSAR.

Vektorski procesor **CLIP-4** je procesor, ki je zasnovan na bitnih rezinah. PE procesorja so razporejene v celično mrežo 96 x 96 PE. Vsaka procesna enota ima osem sosedov v dvodimenzionalnem polju, lahko pa je povezana le s štirimi najbližjimi sosedi.

Računalnik **MPP** je t.im. masivni paralelni procesor (massively parallel procesor), ki je bil razvit v NASA Goddard Space Flight Center za procesiranje satelitskih posnetkov. "Masiven" se imenuje zato, ker ima 128 x 128 = 16384 mikroprocesorjev, ki lahko vsi hkrati paralelno procesirajo. Procesor MPP računa v aritmetiki bitnih rezin z operandi spremenljivih dolžin. Ima mikroprogramljivo krmilno enoto, ki ji je mogoče vprogramirati popolnoma prilagodljive instrukcije za vektorske, skalarne in I/O operacije. Sistem MPP je izveden v celoti v polprevodniški integrirani tehnologiji vezij in uporablja mikroprocesorske čipe in bipolarne RAM-pomnilnike.

WBSPAS. Procesorji tega razreda so sestavljeni iz kombinacije arhitektur strojev iz razreda BPWSAR in WPBSAR. Arhitekturo WBSPAR sestavlja dvodimenzionalni pomnilnk, iz katerega se lahko čitajo ali besede ali bitne rezine, horizontalna procesna enota, ki procesira besede, in vertikalna procesna enota, ki procesira bitne rezine. To je v bistvu ortogonalni računalnik, ki ga je zasnoval Shooman že leta 1970. Oba omenjena računalnika iz prejšnega razreda ICL DAP in STARAN sta sicer lahko programirana tako, da zagotavljata sposobnosti računalnika iz tega razreda, ker pa nimata ločenih procesnih enot za procesiranje besed in bitnih rezin, ne pripadata temu razredu. Implementacijo strojev, katerih arhitektura popolnoma ustreza definiciji strojev iz razreda WBSPAS, predstavlja serija računalnikov **OMEN-60**. Računalnik OMEN-60 je zasnoval Higbie že leta 1972, njihovo arhitekturo pa kaže slika 7.



Slika 7. Arhitektura procesorja iz razreda WBSPAR.

Serija računalnikov **OMEN** (Ortogonal Mini EmbedmeNt) je komercialna implementacija ortogonalnega računalniškega koncepta proizvajalca Sanders Associates za aplikacije, kot je procesiranje signalov. Ta serija uporablja računalnik PDP-11 za konvencionalno horizontalno aritmetično enoto in polje 64 procesnih enot (PE) za asociativno vertikalno aritmetično enoto. Operacije te enote se izvajajo predvsem nad zlogovnimi rezinami in ne toliko nad bitnimi rezinami, odvisno od modela; ali ima le-ta bitno serijsko aritmetiko z osmimi biti pomnilnika, ki pripada vsaki PE, ali pa ima aritmetiko v plavajoči vejici, izvedeno v materialni opremi z osmimi 16-bitnimi registri in petimi maskovnimi registri. Posebna logika med PE-mi obrača vrstni red zlogov v rezini ali izvršuje popolno premeščanje (perfect shuffle) ali ciklični pomik.

Med MISMD primere tega razreda lahko prištevamo računalnik **MAP** (Multi-Associative Procesor) z osmimi CU, ki jim je dodeljeno 1024 PE in sistem PM^4, ki je bil načrtovan kot "rekonfigurabilni" računalniški sistem. Posebnost tega procesorja je, da lahko procesira na tri načine: kot MSIMD, MSISD in kot MIMD. Tipično konfiguracijo PM^4 sestavlja 16 centralnih enot s 1024 procesorskimi pomnilniškimi enotami. Sistem je bil kasneje na Pardue University izpopolnjen v t.im. **PUMPS** tako, da ga je mogoče uporabljati tudi za splošne raziskave večprocesorskih sistemov.

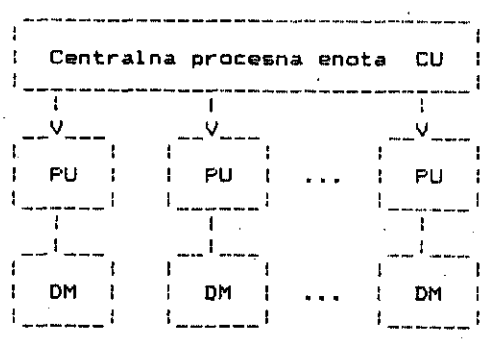
BPWSAS. Arhitektura računalnikov tega razreda je analogna arhitekturi računalnikov iz razreda WPBSAR, kot jo kaže slika 5. V ta razred spadajo asociativni procesorji, ki uporabljajo besedno serijski asociativen pomnilnik. Na tovrstnih procesorjih je zasnovan tudi računalnik PEPE. Ker je le-ta hkrati tudi t.im. "nepovezan asociativen procesor", ga bomo obravnavali v prav tako imenovanem razredu UNCNAS.

WPBSAS. Arhitektura računalnikov tega razreda je analogna arhitekturi računalnikov iz razreda WPBSAR (Slika 6). Predstavnik tega razreda je **STARAN**. Uporablja bitno serijski asociativen pomnilnik. Ima do 32 asociativnih vektorskih modulov. Prvi računalnik tega tipa je bil namenjen procesiranju digitaliziranih slik leta 1975. Popolnoma paralelna struktura uporablja zahtevno in drago logiko v vsaki pomnilniški celici in zapletene komunikacije med celicami. Bitno serijski asociativen procesor je bistveno

cenejši od popolnoma paralelne strukture, ker se istočasno primerja le ena sama bitna rezina. Vsak od 32 asociativnih vektorskih modulov ima 256 x 256 bitnih besed večdimenzionalno dostopnega pomnilnika (MDA), 256 procesnih elementov, permutacijsko (flip) vezje in selektor. Vsak procesni element obravnava serijsko bit po bitu nad podatki v vseh besedah MDA pomnilnika. S pomočjo permutacijskega vezja so podatki, ki so shranjeni v pomnilniku DMA, dostopni preko I/O kanalov v bitnih ali besednih rezinah ali v kombinaciji obeh. Permutacijsko vezje izvaja pomike ali opravila, ki omogočajo paralelno iskanje, aritmetične ali logične operacije nad besedami pomnilnika MDA. Ta pomnilnik je Goodyear Aerospace implementiral tako, da je RAM čipom dodal XOR logična vezja, v kasnejših, dražjih modelih pa je MDA-pomnilnik povečal na 9216 x 256 bitov na modul in znatno povišal hitrost I/O operacij in procesiranja. Med izvajanjem ene instrukcije se podatki v vseh selektiranih pomnilnikih vseh modulov procesirajo sočasno s preprostimi procesnimi elementi, ki so dodeljeni vsaki podatkovni besedi. Vmesniška enota vsebuje vmesnike s senzorji, konvencionalne računalnike, signal procesorje, interaktivne prikazovalnike in masovne pomnilniške naprave. Različne I/O opcije so implementirane v "custom design" vmesniški enoti, ki omogoča direkten dostop do pomnilnika, I/O kanale z vmesnim shranjevanjem, zunanje funkcijske kanale in paralelni I/O. Vsak asociativni vektorski modul ima 256 vhodov in 256 izhodov, prav tako v posebni "custom design" vmesniški enoti. Ta omogoča večjo hitrost komunikacij vektorskih podatkov računalnika z visoko pasovno širino I/O naprave in dovoljuje katerikoli napravi, da neposredno komunicira z asociativnimi vektorskimi moduli. Navedene sposobnosti znatno povečujejo v zahtevnih aplikacijah prepustnost sistema, poenostavljeno kompleksnost programske opreme pri sorazmeroma manjši ceni materialne opreme. Posebna odlika računalnika STARAN je visoka I/O hitrost oz. zmogljivost komuniciranja z okolico in sposobnost preprostega povezovanja z konvencionalnimi računalniki. STARAN v sistemu s konvencionalnimi računalniki nadzoruje paralelno procesiranje opravil, konvencionalni računalniki pa opravila, ki se morajo procesirati serijsko.

Dandanes je večina asociativnih procesorjev načrtovanih za iskanje informacij in obdelave podatkovnih baz. Takšne vrste je asociativen računalnik **RELACS**, ki ga je predlagal Stevens (1979) in je namenjen raziskavam na Syracuse University. Zasnovan je na uporabi večstopenjskih pomnilnikov med diski in gostiteljskim računalnikom.

UNCNAS. Stroji tega razreda imajo pomnožene PU in DM iz strojev prvega razreda BPWSAR. PU in pridruženi DM so v tem razredu definirani kot



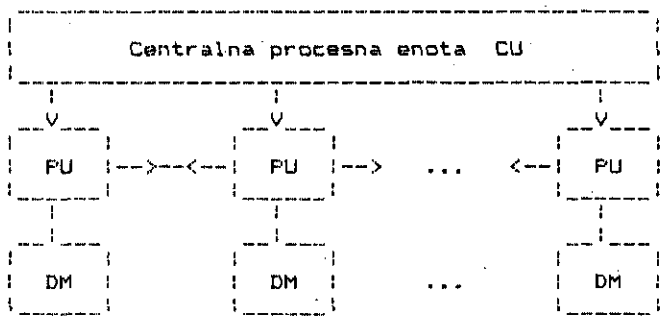
Slika 8. Arhitektura procesorja iz razreda UNCNAS.

procesni elementi PE. Vse PE izvajajo instrukcije ene same krmilne enote CU. Dobro poznan stroj tega razreda je PEPE. Ker procesne enote med seboj niso povezane, je uporaba teh strojev omejena, po drugi strani pa obstajajo določene prednosti le-teh zaradi relativno enostavnih procesnih elementov. Arhitekturo računalnikov tega razreda kaže slika 8.

Kot vidnejšega predstavnika tega razreda si bomo ogledali računalnik **PEPE**.

Obstajata dva tipa popolnoma paralelnih asociativnih procesorjev: besedno organizirani in s porazdeljeno logiko. Pri besedno organiziranih procesorjih je prisotna primerjava vsakega bita vsake besede in so možne logične odločitve po procesiranju vsake besede. Asociativni procesorji s porazdeljeno logiko so manj zapleteni in zato tudi cenejši. Takšen procesor je PEPE, ki je bil razvit v Bell Laboratory-jih za aplikacije procesiranja radarskih signalov. Procesor sestavlja sedem različnih funkcionalnih podsistemov. Od teh je pet nadzornih enot, kot so nadzorna enota izhodnih podatkov in pomnilniških elementov, aritmetična, korelacijska in asociativna izhodna nadzorna enota, ter nadzorni sistem in številne procesne enote. Vsako PE sestavlja aritmetična enota, korelacijska enota, asociativna izhodna nadzorna enota in pomnilnik z 1024 32-bitnimi besedami. 288 PE je organiziranih v osem elementne svitke. Selektirani deli opravil se nalagajo iz gostiteljskega računalnika CDC-7600 v procesne enote. Selekcijo določa inherentni paralelizem opravil in specifična arhitektura procesorja PEPE, ki lahko obravnava posamezna opravila veliko bolj učinkovito kot gostiteljski univerzalni računalnik, in se uporablja le kot koprocesor tega računalnika.

CNAR. Stroji tega razreda so podobni strojem iz prejšnega razreda s to razliko, da so pri strojih iz razreda CNAR procesne enote PE razvrščene v liniji in med seboj povezane znajbližji sosedi. To pomeni, da morejo nekatere procesne enote naslavljati besede v svojem lastnem pomnilniku in tudi od neposrednih sosedov. Na ta način je v določeni meri odpravljena slabost procesorjev iz prejšnega razreda, ki izvira iz nepovezanosti PE. Primer stroja tega razreda je **ILLIAC IV**, ki omogoča hkrati tudi direktne komunikacije med vsakimi osmimi procesnimi enotami PE. Karakteristično shemo arhitekture procesorja iz tega razreda vidimo na sliki 9.



Slika 9. Arhitektura procesorja iz razreda CNAR.

Vidnejši predstavniki tega razreda so poleg računalnika ILLIAC IV tudi izvedenke MIMD tega računalnika, kot je procesor **PHOENIX**. Njegov

predhodnik je procesor, ki ga v bistvu sestavlja originalen računalnik ILLIAC-IV. Ta je načrtovan tako, da ima štiri centralne enote, ki jim je dodeljeno 256 PE (4 krat 64 PE). V okviru računalniškega projekta PHOENIX pa je združenih 16 po 64 PE, kot razširitev računalnika ILLIAC IV za še učinkovitejše MSIMD vektorsko procesiranje.

Med procesorje MIMD tega razreda lahko prištevamo tudi **TRANSPUTERSKI SISTEM**, ki ga bomo posebej obravnavali kasneje.

Pregled zmogljivosti nekaterih paralelnih računalnikov daje Tabela 1.

SIMD sistemi (AR/AS)	Leto zasnove	Arhitektura	Največja zmogljivost v Mflop/s
UNGER	1958	ws,AR	
SOLOMON	1962	ws,AR	
VAMP	1965	ws,AR	
CDC 7600	1969	ws,AR	
STARAN	1970	bs,AS	
ILLIAC IV	1972	ws,AR	80-200
PEPE	1973	bs,AS,UNCN	100-288
CLIP	1967	bs,AR	10^8 pixel ops
OMEN 60	1976	bs,AS,ORT	
MAP	1977	ws,AS,M	
GRAY-1	1978	ws,AR *	130-200
ICL DAP	1978	bs,AR	10-30
BSP	1979	ws,AR	20-50
PM4	1979	ws,AS,M	
RELACS	1979	bs,AS	
PHOENIX	1979	ws,AR,CN,M	10^4
CDC NASF	1979	ws,AR *	1000-3000
CYBER 205	1982	ws,AR	3000
MPP	1983		200-6000

Tabela 1. SIMD-računalniški sistemi

Legenda:

ws	besedne rezine	M(SIMD)	večkratni (Multiple SIMD)
bs	bitne rezine		
AR	vektorski	ORT	ortogonalni
AS	asociativni	CN	povezani
*	SISD z MFE	UNCN	nepovezani

2.2. Pregled razvoja paralelnih sistemov.

V Tabeli 2 je prikazan časovni prikaz razvoja paralelnih vektorskih in asociativnih procesorjev. Med prve , ki so jih zasnovali, sodijo: von Neumann, Holland in Shooman. Povezave med posameznimi procesorji v razpredelnici kažejo v smeri od desne proti levi njihove prednike po arhitekturnih značilnosti.

V zaključku tega sestavka naj omenimo, da so stroji MIMD neprimerni za reševanje problemov, ki se nanašajo na reševanje sistemov parcialnih diferencialnih enačb (p.d.e.s.) ali razsežnih optimizacijskih problemov. Tako nam za reševanje tovrstnih problemov ostanejo le matrični in vektorsko cevani računalniki. Med prve sodi na primer ICL DAP, med druge pa Gray-1. Od obeh je za omenjene raziskave primernejši vektorski računalnik ICL DAP iz naslednji razlogov:

1. Mreža procesorjev (matrika) se enostavno preslika v mrežo, ki je za tip problemov, ki se rešujejo, najustreznejša, in
2. Vektorski računalnik omogoča poglobljen študij sočasnosti v algoritmu, medtem ko vektorsko cevani računalnik (npr. GRAY-1) sočasnost (paralelizem) v algoritmu na nek način prekriva.

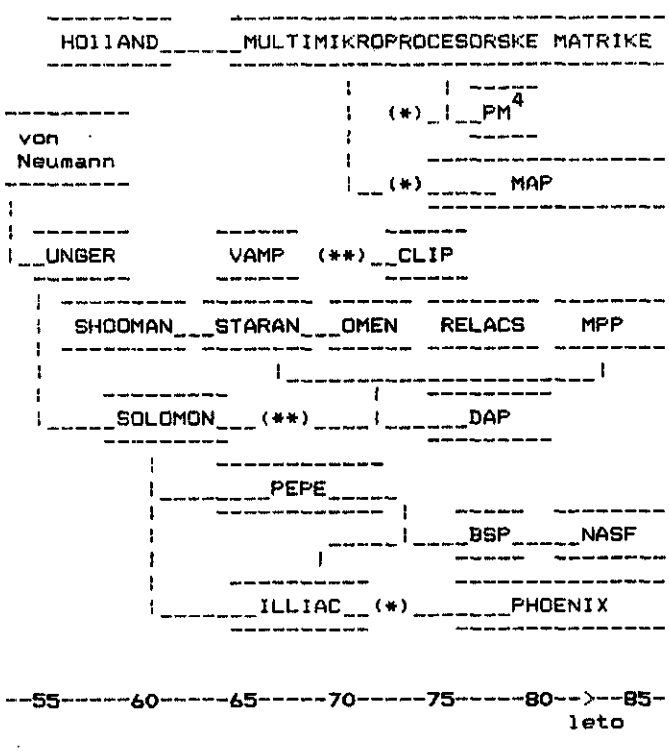


Tabela 2. Razvoj paralelnih vektorskih in asociativnih procesorjev.

Zato se bomo v naslednjem poglavju poglobili v strukturo vektorskega računalnika, kakršna sta na primer računalnika ICL DAP z jezikom DAP FORTRAN in novejši TRANSPUTERSKI SISTEM z jezikom OCCAM.

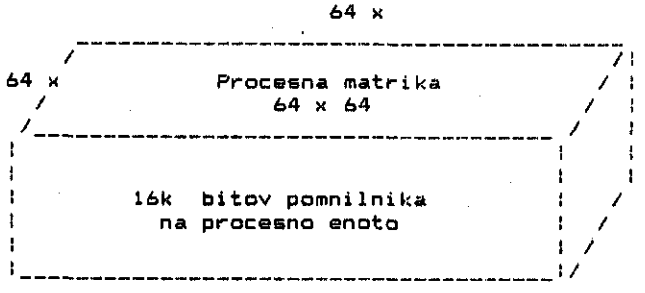
3. PORAZDELJEN VEKTORSKI RAČUNALNIK

Zgled vektorskega porazdeljenega procesorja je računalnik ICL DAP. Med novejše tovrstne računalnike lahko prištevamo tudi sistem t.im. transputerjev. Transputer predstavlja procesorski modul. Vektorski računalnik pa dobimo, če te module matrično povežemo med seboj.

Ena od bistvenih razlik obeh računalnikov je v tem, da ICL DAP pripada razredu SIMD, medtem ko lahko TRANSPuterski Sistem pripada tudi razredu MIMD. Prvi ima v današnji verziji (Queen Mary Colege, University of London) v enoti 4096 procesorjev, ki so razporejeni v mariki 64 x 64. Ta enota ne dela kot samostojni računalnik, ampak le kot pomnilniški modul, ki je povezan z običajnim centralnim pomnilnikom. Podrobnejši opis računalnika DAP najdemo v delu Reddaway-a /8/. V TRASPS pa so elementi matrike povsem samostojni procesorji tim. TRANSPUTERJI, ki pripadajo procesorjem iz razreda SISD /9/. Ne le v omenjenem primeru, ampak tudi sicer lahko ugotavljamo, da imajo procesorji v računalnikih, ki pripadajo računalnikom iz razreda SIMD, v primerjavi s procesorji v računalnikih, ki pripadajo razredu MIMD ali centralnim procesorjem, praviloma zelo preprosto osnovno arhitekturo. V računalniku DAP so procesorji, ali bolje rečeno procesne enote PE, bitno organizirane, kar omogoča sistemu veliko prilagodljivost aplikacijam, ki so skladne z njegovo organizacijo. Med takšne, najbolj pogoste aplikacije, sodijo: procesiranje slik, pregledovanje podatkov in simbolno procesiranje, ki je združeno z osnovno aritmetiko.

3.1. Računalnik ICL DAP

Osnovno organizacijsko shemo računalnika DAP podaja slika 10. Iz nje je razvidno, da je vsaka procesna enota povezana z najbližjimi štirimi sosedi. Glavne podatkove poti, ki potekajo po kolonah in vrsticah, pa povezujejo vse procesorje dane vrstice oziroma kolone. Te poti omogočajo hitro oddajanje in sprejemanje informacij preko matrike procesorjev in je mogoče vzpostavljati diametralne povezave, tudi preko celotne matrike, ali več ločenih povezav. V računalniku DAP ima vsak procesor 16k bitov pomnilnika RAM, kar predstavlja skupaj 16k x 64 x 64 / 8 = 8 megazlogov. Pri tem naj omenimo, da imajo drugi podobni stroji, ki so danes instalirani, le 4 kilozlogov na procesor ali skupaj 2 megazloga.

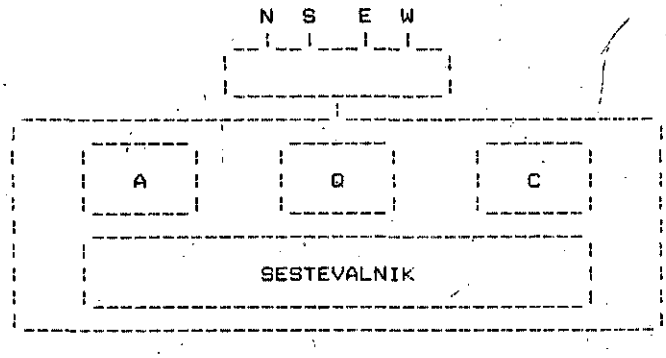


Sika 10. Osnovna organizacijska shema računalnika ICL DAP

Vsaka procesna enota poseduje svoje lastne podatke, ki jih obdeluje, sprejema pa tudi nekaj skupnih instrukcij, ki jih pošilja glavna kontrolna enota (MCU). Zato ta računalnik prištevamo med računalnike tipa SIMD.

Iz slike 11 je razvidno, da ima procesna enota PE tri registre: delovni register A omogoča prekinjajoče delovanje procesorja v skladu z zahtevami programa, medtem ko je A enobitni akumulator in C register, ki hrani prenos.

Pomembna lastnost računalnika DAP je sposobnost izključevanja posameznih procesorjev v računalniku. Na primer, če rešujemo problem, za reševanje katerega zadošča mreža procesorjev, ki je manjša od 64 x 64, moremo s pomočjo logične matrike odvečne procesorje z maskiranjem preprosto izključiti. Na ta način moremo izvajati



Slika 11. Arhitektura procesne enote procesorja ICL DAP.

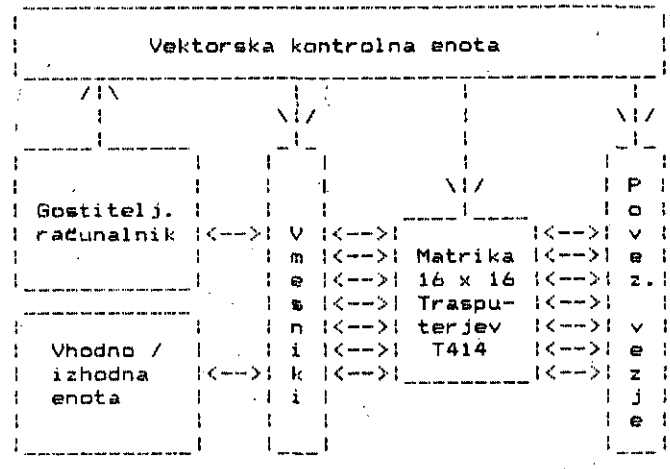
neke matematične operacije na delu matrike, ki je prilagojena določenim zahtevam. Slednje si bomo kasneje ogledali bolj nadrobno.

3.2. SISTEM TRANSPUTERJEV

Računalniki z več med seboj povezanimi procesnimi enotami, imenovanimi TRANSPUTRJI /12,13, 14/, predstavljajo različne sisteme, med katerimi bo za nas posebej zanimiv porazdeljen vektorski računalnik.

Inmos, ena najpomembnejših britanskih firm računalniških komponent, je ponudila tržišču 16-bitne (T 212 in M 212) in 32-bitne (T 414), napoveduje pa tudi že nove še zmogljivejše (T 800) RISC procesorske enote, imenovane transputerje, ki so med seboj popolnoma združljive. Transputer je računalnik na čip, ki omogoča izvajanje več procesov hkrati in tudi sam skrbi za komunikacijo med njimi. Komunikacija poteka preko skupnega pomnilnika. Več transputerjev se lahko povezuje med seboj preko kanalov v večprocesorski sistem, ki omogoča konkurenčno izvajanje večih procesov. Takšne sisteme je moč še nadalje povezovati v še večje sisteme in tako graditi sisteme s poljubnim številom transputerjev. Transputerski sistemi niso več zasnovani na von Neumannovi arhitekturi, kakršno ima sam transputer. Zato, in zaradi sposobnega transputerja lahko dosegajo ali celo presegajo zmogljivost današnjih superračunalnikov. Zaključeni transputerski sistemi se povezujejo med seboj in s standardno mikroprocesorsko periferijo preko posebnih vmesnikov t.im. "link adaptorji" IMS C001 in IMS C002, ki skrbijo za medsebojno sinhronizacijo večih sistemov oziroma povezavo sistema z njegovo periferijo.

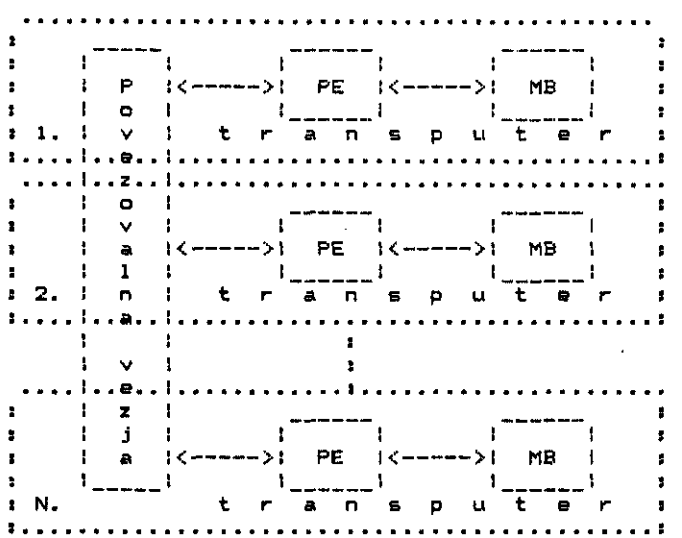
32-bitni transputer T 414 je splošno namenski in zmore 10 MIPSov pri 20 MHz. Prav tako je splošnonamenski njegov predhodnik T 212, medtem ko je M 212 namenski transputer za kontrolo inteligentnega diskovnega sistema. T 414, ki je predstavnik te družine, je izdelan v 1.5 mikronski CMOS tehnologiji s preko 150k transistorjev v 84-pinskem čipu. Procesor ima 32-bitne notranje in zunanje izhode za naslove in podatke, ki so multipleksirani in dosegajo hitrost prenosa 20 megazlogov, 4-gigazložni linearni naslovni prostor, PROM in 2k SRAM pomnilnika ter 4 medtransputerske komunikacijske kanale. V pogledu nabora ukazov transputer odstopa od običajnega nabora ukazov RISC arhitekture, predvsem po številu vseh ukazov in prisotnosti ukazov za množenje in deljenje.



Slika 12. Transputerski vektorski sistem

Slika 12 kaže transputerski vektorski sistem, ki ga sestavljaja kvadratna matrika N=16x16 procesnih enot, sistem pomnilniških vmesnikov, povezovalni sistem, matrična krmilna enota, gostiteljski računalnik in vhodno/izhodna enota.

Kot smo že omenili, so transputerji zelo zmogljivi 32 bitni procesorji (n.pr. T424)s statičnim pomnilnikom (pomnilniško banko) in različne učinkovitimi komunikacijskimi vmesniki. Vse te



Slika 13. Konfiguracija komunikacij znotraj vektorskega polja (PE=procesne enote, MB=pomnilniški bloki, N=16x16)

komponente so integrirane v enem samem čipu, ki predstavljajo odlične gradnike konkurenčnih procesnih vezij /15/. Transputerske zanke predstavljajo kanale za medprocesne komunikacije v materialni opremi. Tako obstaja zelo tesna povezanost med povezavami (zankami) transputerskih kanalov in komunikacijskim protokolom, ki je primeren za komunikacije v t.im. "wavefront" vektorskih procesorjih (wavefront array processors).

Proizvajalec transputerjev je poskrbel tudi za učinkovito in lahko programiranje v jeziku OCCAM, ki ga je Inmos posebej razvil za transputer. Prevedli so ga že tudi za druga okolja, npr. VAX in IBM PC.

4. PROGRAMIRANJE VEKTORSKEGA RAČUNALNIKA

Najpreje si bomo ogledali, kakšno je programiranje na paralelnem vektorskem računalniku DAP, medtem ko si bomo programiranje na TRANSPUTERSKEM SISTEMU ogledali v nadaljevanju tega članka.

Računalnik DAP na Univerzi v Londonu je programljiv v jeziku DAP FORTRAN. Ta jezik je standardni FORTRAN, ki pa je razširjen za vektorsko procesiranje. Opis jezika DAP FORTRAN najdemo v delu Flandersa /9/ ali v originalnih priročnikih računalnika ICL DAP.

V standardnem FORTRANu je osnovna računska enota skalar, v DAP FORTRANu pa je le-ta lahko skalar, vektor ali matrika (kjer so vektorji in matrike dimenzije N oziroma N x N. Pri računalniku DAP je dimenzija N = 64). Medtem, ko lahko seštevanje dveh matrik v običajnem FORTRANu zapišemo kot

```
      DO 10 I=1,N
      DO 10 J=1,N
          C(I,J)=A(I,J)+B(I,J)
10    CONTINUE,
```

bo v DAP FORTRAN gornji zapis preprosto skrčen na en sam stavek

```
      C=A+B
```

Jezik DAP FORTRAN ima številne osnovne funkcije, ki se uporabljajo za izvajanje določenih operacij. Takšne funkcije so na primer:

SUM Izračuna skalarno vsoto matrike N x N.

ABS Zapiše absolutno vrednosti vsakega elementa v matriki N x N.

MAXV Poišče največje skalarne vrednosti v matriki N x N.

MAXP Poišče pozicijo največje vrednosti (oziroma največjih vrednosti, če jih je več) v matriki N x N. Pozicija je označena z "1" v logični matriki.

ALL Izračuna logično vsoto IN vseh vrednosti v logični matriki (ali logični izraz, ki izračuna logično matriko). Funkcija je izredno učinkovita pri konvergenčnem testu matrike.

Pomemben pripomoček pri obravnavanju matrik predstavlja, kot smo že omenili, sposobnost izklapljanja posameznih procesorjev s pomočjo logične matrike. V nadaljevanju bomo pokazali nekaj primerov.

Predpostavimo, da imamo matriko A razsežnosti 64 x 64, ki vsebuje pozitivna in negativna števila. Iščemo kvadratni koren vseh pozitivnih elementov matrike. To lahko zapišemo v običajnem FORTRANu takole:

```
      DO 10 I=1,N
      DO 10 J=1,N
          IF(A(I,J)GT.0)=SQRT(A(I,J))
10    CONTINUE
```

medtem ko je v DAP FORTRANu zgornji program bistveno preprostejši:

```
      A(A.GT.0)=SQRT(A)
```

Izraz A.GT.0 predstavlja logično matriko z vrednostmi "1" na pozicijah, kjer je A večji od 0, drugod pa vrednost "0". Rezultat na desni strani izraza je matrika, kateri je dodeljena vrednost le na tisti poziciji, ki ji ustreza v logični matriki na levi strani vrednost "1". Oglejmo si bolj zapleten primer, kot je reševanje Laplace'ove enačbe. V tem primeru želimo zamenjati vsako vrednost v matriki s povprečno vrednostjo njenih štirih najbližjih sosedov. V standardnem FORTRANu bomo zapisali tonamensko kodo takole:

```
      DO 10 I=2,N-1
      DO 10 J=2,N-1
          Y(I,J)=(X(I+1,J)+X(I-1,J)+X(I,J+1)+
                  X(I,J-1))/4
10    CONTINUE
```

Pri tem je potrebno zapisati posebno kodo, ki ureja meje matrike (tj. preprečuje napake, ki bi se lahko pojavile zaradi matričnih indeksov, ki presegajo meje matrike). Kodo zgoraj lahko v DAP FORTRANu izrazimo z enim samim stavkom

```
      X=(X(+,)+X(-,)+X(,+)+X(,-))/4
```

V slednjem izrazu izkoriščamo sposobnost računalnika DAP, ki omogoča pomično indeksiranje. Z izrazom X(+,) dosežemo točko na sosednji vrstici matrike procesnih enot (ki ustreza izrazu X(I+i,J) v serijski verziji), z X(-,) pa dosežemo točko v predhodni vrstici, itd. Pomembno je, da vemo, da so vse točke v matriki istočasno ažurirane. V takšnem primeru seštevalne matrike ne potrebujemo več. Meje matrike se urejajo avtomatično z vnašanjem ničel na mesta, ki jih določa geometrija. Pri ravninski geometriji vstavljamo ničle na ustrezna mesta v ravnini, medtem ko pri cikličnj geometriji zato, da dobimo cilinder, povezujemo ali severjužne ali vzhod-zahodne robove, neodvisno od uporabljene smeri pomika. Lahko pa pomikamo tudi vse štiri robove in s tem dobimo "torus".

Sedaj predpostavimo, da želimo rešiti Laplaceov problem z iregularno oblikovano mejo (ali s pravokotnim poljem, ki je manjše od 64 x 64). DAP FORTRAN omogoča kreiranje logične matrike, imenovane DOMAIN. V njej označimo pravilne vrednosti (TRUE: logična "1") v tiste dele matrike, ki odgovarjajo področju problema in neprave vrednosti (FALSH: logična "0") povsod drugod v matriki. Rešitev je podana v naslednji kodi:

```
      DO 10 I=I,LIMIT
           OLDX=X
           X(DOMAIN)=(X(+,)+X(-,)+X(,+)+X(,-))/4
      IF(ALL(ABS(X-OLDX).LT.EPS)) GOTO 20
10    CONTINUE
      CONVERGED=.FALSE.
      RETURN
20    CONVERGED=.TRUE.
      RETURN
```

5. ZAKLJUČEK

S tem smo se na kratko seznanili z naravo računalnika ICL DAP in načinom programiranja, ki je primeren za takšen računalnik. Iz teh spoznanj moremo sklepati, da je učinkovitost računalnika ICL DAP za reševanje tako linearnih kot nelinearnih p.d.e. izredno velika. To dejstvo je prepričljivo potrjeno tudi s strani uporabnikov tega računalnika.

Drugi del prispevka bo posvečen programiranju TRANSPUTERSKEGA SISTEMA in optimizacijskemu postopku, ki je zasnovan na matričnem računu in ga je mogoče učinkovito izvajati le na paralelnem vektorskem računalniku. Omenjeni postopek se lahko uporablja kot učinkovito orodje za implementacijo uporabniških algoritmov, ki vsebujejo veliko stopnjo inherentnega paralelizma, na paralelnem vektorskem računalniku.

6. REFERENCE

/1/ Flynn M., IEEE Transactions on Computers, C-21,9, pp. 948 - 960, 1972.

/2/ Barlow R.H., The Neptun Processing System, Loughborough University of Technology.

/3/ Johnson D.,et al., Automatinc Partitioning of Programs in a Multiprocessor System, Texas Instruments, Austin, Texas, 1979.

/4/ Sauber W., A Dataflow Architecture Implementation, Texas Instruments, Austin, Texas, 1980.

/5/ da-Silva J.G.D., Woods J.V., Design of a processing subsystem for the Manchester data-flow computer, Proceedings of the IEEE, 128,5, 1981.

/6/ Hwang K., Briggs F.A., Computer Architecture and Parallel Processing, McGraw-Hill Book Company, 1985.

/7/ Kogge P.M., The Architecture of Pipelined Computers, McGraw-Hill Book Company, 1981.

/8/ Robič B., Silc J., Razvrstitev novogeneracijskih računalniških arhitektur, Informatica 10,4, 1986.

/9/ Hockney R.W., Jesshope C.R., Parallel Computers, Adam Hilger Ltd, Bristol, 1981.

/10/ Foster C.C., Content Addresable Parallel Processors, Van Nostrand Reinhold Company, 1976.

/11/ Hsiao D.K., Advanced Data Base Machine Architecture, Prentice-Hall, Englewood Cliffs, N.Y.,1983.

/12/ INMOS Limited, OCCAM language overview, November 1985.

/13/ INMOS Limited, Transputer architecture, November 1985.

/14/ INMOS Limited, IMS T414 Transputer, November 1985.

/15/ MihovilovićB., Mavrič S., Kolbezen P., Transputer - osnovni gradnik večprocesorskih sistemov, Informatica 4/81, 1986.

Vido Vouk
Jure Ferbežar
Andrej Brodnik
Institut »Jožef Stefan«

UDK 681.3.014

This article presents a multi tasking environment for the MS-DOS operating system on the IBM-PC computer. Real time scheduler was developed using real time clock interrupts to perform process scheduling. Interprocess communication is implemented using semaphores and message exchange. All the support routines for the multi-tasking real-time environment are packed in a single module which offers all the routines needed to handle process manipulation, process synchronization and interprocess data exchange. All the software is developed using Logitech Modula-2 environment.

Članek opisuje osnovno okolje za pisanje večopravilnih programov na računalniku IBM-PC pod operacijskim sistemom MS-DOS. V ta namen smo razvili razporejevalnik procesorskega časa, ki uporablja urine prekinitve. Za medprocesno sinhronizacijo smo uporabili semaforje in izmenjavo sporočil. Uporabniku je na voljo modul podprogramov, ki nudi vse potrebne podprograme za delo v večprocesorskemu okolju. V modulu so podprogrami za ustvarjanje, poganjanje, ustavljanje in izločanje ter podprogrami za nadzor procesov, podprogrami za komunikacijo med procesi (send, receive) in podprogrami za sinhronizacijo (wait, signal) in ustrezni dodatni podprogrami za ustvarjanje, nadzor in izločanje semaforjev. Vsa programska oprema je napisana v programskem jeziku modula-2.

1. UVOD

Računalniki združljivi z IBM-PC so pri nas vedno pogostejši in tudi niso pretirano dragi. Žal pa ti računalniki ne nudijo podpore v večopravilnem okolju. Ker se zaradi nizke cene vedno več potrošnikov odloča za ta tip računalnika, so razvijalci prisiljeni poiskati, oziroma izdelati čimbolj univerzalna orodja za izdelavo zahtevnejših aplikacij. V tem članku predstavljeni razporejevalnik procesorskega časa predstavlja eno od takih orodij. Žal zaradi omejenosti operacijskega sistema MS-DOS iz fička ni mogoče narediti Ferrarija.

1.1 PROCESI

Proces lahko definiramo kot asinhrono aktivnost, naprimer izvajanje programa na centralni procesorski enoti CPE /HOL78/. S preprostim razmislekom lahko pridemo do zaključka, da je proces lahko v poljubnem trenutku opazovanja le v enem od dveh možnih stanj

IZVRŠLJIV ---------- NEIZVRŠLJIV

Opisani stanji lahko zaradi lažjega razmišljanja razdelimo naprej na naslednja podstanja:

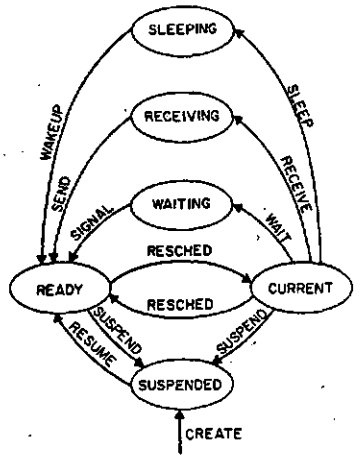
IZVRŠLJIV	- TRENUTNI
	- PRIPRAVLJEN
NEIZVRŠLJIV	- PREKINJEN
	- ČAKAJOČ
	- SPEČ
	- SPREJEMAJOČ

Privzemimo, da imamo več procesov. Vsi procesi iz skupine IZVRŠLJIV imajo pravico do izvajanja na poljubnem CPE. V poljubnem trenutku pa se dejansko izvaja eden (na eno procesorskem sistemu) ali N procesov (na N procesorskem sistemu). Procese, ki se izvajajo, imenujemo TRENUTNI. Vsi ostali procesi, ki imajo pravico do izvajanja, pa se ne izvajajo zaradi pomanjkanja prostih procesorjev, so v stanju PRIPRAVLJEN.

Procesi iz skupine NEIZVRŠLJIV so začasno ustavljeni in nimajo pravice do procesorja, čeprav bi kak procesor bil prost ali celo brez dela. Ti procesi so ustavljeni in čakajo na nek zunanji dogodek, ki jih zbudi in jih prestavi v skupino IZVRŠLJIVIH procesov. Dogodki, ki jih zbudijo in prestavijo v drugo skupino, so različni glede na stanje v katerem so zaustavljeni.
Slika 1.1 prikazuje možna stanja procesov in dogodke, ki vplivajo na prehode med stanji.
K sliki 1.1 moramo dodati tudi kratek komentar. Iz slike je razvidno, da klic podprograma CREATE ustvari nov proces in ga postavi v stanje PREKINJEN. Vprašanje, ki se pojavi je, kaj se zgodi s procesom, ki ga želimo odstraniti iz našega okolja. Klic KILL odstrani proces ne glede na to kje se je nahajal pred klicem. Proces nepreklicno izgine iz okolja. Pred odstranitvijo sprosti vse zasedene zmogljivosti v sistemu.

1.2 SINHRONIZACIJA

Za sinhronizacijo procesov v večprocesornih okoljih uporabljamo različne tehnike. Dokažemo lahko, da so vse tehnike funkcionalno enakovredne /FIL84/, vendar so v



Slika 1.1
Stanja procesov in prehodi med stanji

različnih izvedbah različno primerne. Razporejevalnik opisan v tem članku za sinhronizacijo uporablja semaforje /DIJ75/ in izmenjavo sporočil /JON87/.

1.2.1 SEMAFORJI

Kot sta pokazala /DIJ75/ in /HAN73/ lahko vse probleme sinhronizacije rešimo s P in V semaforjema. V naši izvedbi razporejevalnika smo uporabili splošne (števne) semaforje. Torej je binarni semafor v našem razporejevalniku le posebna oblika splošnega semaforja.

1.2.2 IZMENJAVA SPOROČIL

Izmenjava sprocil (message exchange) je poseben mehanizem, ki omogoča procesu da prenese pripravljeno sporočilo drugemu procesu. Izmenjava sporočil je obenem poseben mehanizem sinhronizacije. Glavna razlika med semaforji in izmenjavo sporočil je v tem, da mora biti za vsak klic WAIT(sem) ustrezen klic SIGNAL(sem), v primeru izmenjave sporočil pa to ni nujno. Izmenjava sporočil je za sinhronizacijo preprostejša posebno kadar proces ne ve vnaprej koliko sporočil bo dobil in kateri procesi jih bodo poslali. Ta način sinhronizacije je uporabljen v CSP /HOA85/, moduli-2 /WIR85/ in v programskem jeziku OCCAM /JON87/. Dokažmo lahko, da je možno izvesti izmenjavo sporočil samo s semaforji in obratno /FIL84/.

1.3 RAZPOREJANJE PROCESOV

Naša nadaljna razlaga bo zadevala enoprocesorske sisteme. Posplošitev na večprocesorske sisteme je očitna.
Iluzijo vzporednega izvajanja več procesov na eno ali večprocesorskemu sistemu lahko dosežemo s preklapljanjem procesorja (procesorjev) med več procesi.
če hočemo zagotoviti, da bodo procesi enakopravno izkoriščali dane procesorske zmogljivosti v sistemu, moramo uvesti razsodnika, ki odloča, kdaj bo kateri od procesov začel z izvajanjem, kdaj bo izvajal svojo kodo in kdaj bo prenehal z delom, da bi prepustil procesor naslednjemu uporabniku. Ta razsodnik je razporejevalnik procesorskega časa. Razporejevalnik odloča kateri proces lahko zasede procesor in ga potem tudi lahko prekine, da dodeli procesor drugemu procesu. Vsakemu procesu določimo nek določen čas, ko lahko teče, ne da bi ga prekinili. Po izteku tega časa damo v izvajanje na tem procesorju nek drug proces.

2. IZVEDBA

2.1 PROGRAMSKO OKOLJE

Razporejevalnik časa smo razvili za IBM-PC združljiv računalnik. Za modulo-2 smo se odločili, ker je višji programski jezik /WIR85/, ki ima vse potrebne konstrukte za delo na strojnem nivoju in obenem osnovno podporo za delo z več procesi. Tako je razvoj in vzdrževanje programske opreme relativno preprosto. Vsa programska oprema razporejevalnika je napisana v Logitech Moduli-2/86 pod operacijskim sistemom MS-DOS 3.20.

2.2 UVOD

Razporejevalnik je zasnovan na primeru razporejevalnika za RT-11 /BRO87/. Za razliko podobnih razporejevalnikov, ki delujejo sinhrono /COL87/, naš razporejevalnik omogoča delo v realnem času. Informacijo o pretečenem času dobi od prekinitev, ki jih generira ura. Urine prekinitve so relativno pogoste. če bi razporejevalnik ob vsaki urini prekinitvi pregledal vse strukture, bi bil odziv sicer izjemno dober, vendar bi bila učinkovitost takega sistema zelo majhna, saj bi se tak sistem večino časa ukvarjal sam s seboj. Tako ob vsaki urini prekinitvi razporejevalnik pregleda le nekatere strukture, medtem ko splošni pregled stanja procesov naredi le na določeno število urinih prekinitev. Uglaševanje izvedemo eksperimentalno.
Razporejevalnik odloča, kateri od procesov bo dobil pravico do izvajanja svoje kode. Razporejevalnik ga tudi prekine med izvajanjem in to tako, da se proces tega ne zaveda. Razporejevalnik shrani vse potrebne podatke o procesu v lokalni pomnilnik, tako da lahko v poljubnem trenutku ponovno zažene prekinjeni proces. Proces nadaljuje z izvajanjem v točki v kateri je bil prekinjen.

Med delom smo naleteli na več težav. Največjo težavo je predstavljala sama zasnova operacijskega sistema. MS-DOS 3.20 je eno uporabniški eno opravilni operacijski sistem, tako da nobeden od sistemskih klicev ni prekinljiv. To težavo smo poskušali zaobiti z uporabo nedokumetirane lastnosti operacijskega sistema /LOG85/, ki s posebno zastavico označi kdaj je v kritičnem odseku. Vendar se ta rešitev med izdatnim testiranjem ni izkazala kot dovolj zanesljiva, ker tudi nekateri podprogrami v moduli-2 niso prekinljivi. Ta problem smo rešili tako, da smo celotni klic DOSa označili kot kritični odsek.

Celoten sistem je sestavljen iz treh funkcionalno ločenih modulov: jedro, semaforji in izmenjava sporočil. Jedro omogoča delo nad procesi (ustvarjanje, prekinitev procesa), modul semaforjev omogoča delo s semaforji (ustvarjanje, SIGNAL, WAIT) modul za izmenjavo sporočil pa omogoča klice SEND in RECEIVE. Ker je naš sistem namenjen tudi uporabi v povsem realnih aplikacijah, so vsi trije moduli skriti v oklepajočem modulu, ki zunanjemu uporabniku onemogoča dostop do kritičnih ukaznih in podatkovnih struktur. Iz modula navzven so izpeljane le funkcije, ki so potrebne za učinkovito uporabo sistema. Uporabnik nima nobene možnosti, da bi z

nepravilno ali nepazljivo uporabo porušil konsistentnost sistema. V dodatku A je podan definicijski modul našega razporejevalnika, iz katerega je razvidno, katere podatkovne in ukazne strukture ter klici so uporabniku na voljo.

2.3 PODATKOVNE STRUKTURE

Za učinkovitejše delo razporejevalnika smo morali dobršen del prekinitvenih podprogramov zamenjati s svojimi novo napisanimi, ki uporabljajo konstrukte iz razporejevalnika. Podatkovne strukture izven modula niso vidne /BRO87/. Osnovne enote, s katerimi uporabnik lahko upravlja, so procesi, semaforji in sporočila. Z njimi operira preko klicev podprogramov in tako vpliva na tek procesov. Vsi podprogrami so monitorji /HAN75/.

2.3.1 PROCESI

Osnovna enota v sistemu je proces. Proces je v moduli-2 izvajanje podprograma brez parametrov /WIR85/ in /LOG86/. V našem sistemu smo definicijo razširili z naslednjimi parametri :

- ime
- interno ime
- življenjski prostor
- prioriteta

Ime je sestavljeno iz niza alfanumeričnih znakov. Namenjeno je le uporabniku. Ob zahtevi za ustvaritev procesa jedro preveri pravilnost parametrov in ustvari proces. Priredi mu interno ime, ki ga vrne uporabniku. Vsi nadaljni klici za delo z ustvarjenim procesom uporabljajo le interno ime. življenski prostor opredeljuje velikost procesu prirejenega delovnega pomnilnika /LOG86/. Prioriteta je pomemben parameter, ki opisuje nujnost izvajanja danega procesa. Večja vrednost predstavlja večjo prioriteto. V osnovni izvedbi našega sistema velja, da se vedno izvaja tisti proces, ki ima najvišjo prioriteto. Kadar je takih procesov več, si ti procesi enakopravno delijo procesor med seboj (round robin). Naslednje izvedbe sistema omogočajo drugačne načine razporejanja /VMS82/.
Vsak proces opredeljuje spremenljivka stanja (prim. slika 1.1). Kot smo omenili, se lahko proces v danem trenutku nahaja le v natanko enem stanju. Kratek opis stanj:

TRENUTNI (CURRENT): Ker je IBM-PC enoprocesorski sistem je lahko v poljubnem trenutku le en TRENUTNI proces. Ta proces izvaja svojo kodo. če sam ne kliče nobenega od podprogramov, ki bi mu lahko spremenil status (WAIT, RECEIVE, SLEEP, SUSPEND), ga prekine razporejevalnik ko poteče njegov interval časa, ali pa če se prebudi kak proces z višjo prioriteto. Kadar je ta proces edini s tako prioriteto, bo ponovno dobival pravico do izvajanja po en interval, dokler ne bo končal ali pa zamenjal statusa.

PRIPRAVLJEN (READY): V tem stanju so vsi procesi, ki imajo vse pogoje za izvajanje in čakajo le na prost procesor.

PREKINJEN (SUSPENDED): Po klicu za ustvaritev razporejevalnik ustvari proces in ga postavi med NEIZVRŠLJIVE procese. Proces ostane v tem stanju, dokler ga ne obudi eden od TRENUTNIH procesov. Razporejevalnik lahko v to stanje postavi tudi poljuben PRIPRAVLJEN proces, če to od njega zahteva kateri izmed procesov.

ČAKAJOČ (WAITING): Razporejevalnik postavi poljuben TRENUTNI proces po klicu WAIT v to stanje, če je vrednost semaforja ob klicu nič. Proces ostane v tem stanju dokler ga s klicem SIGNAL ne obudi eden izmed TRENUTNIH procesov.

SPEČ (SLEEPING): Proces postavi v to stanje razporejevalnik po klicu SLEEP. Iz tega stanja ga spet obudi razporejevalnik po izteku zahtevanega časovnega intervala.

SPREJEMAJOČ (RECEIVING): Po klicu RECEIVE proces preide v to stanje, če zanj še ni prispelo nobeno sporočilo in ostane v tem stanju dokler ne pride sporočilo od enega izmed TRENUTNIH procesov.

2.3.2 SEMAFORJI

Semaforji v našem sistemu predstavljajo osnovni princip sinhronizacije. Uporaba in delovanje so obširno opisani v literaturi (npr. /HAN73/). V naši izvedbi razporejevalnika uporabljamo splošne semaforje. Tip semaforja določimo s klicem podprograma za ustvaritev semaforja.
Vsak klic SIGNAL (P) poveča vrednost semaforja za 1 če je vrsta čakajočih na ta semafor prazna, sicer pa za vsak SIGNAL obudi enega od čakajočih.
Po vsakem klicu WAIT (V) razporejevalnik pogleda ali je števec semaforja večji od nič. če ni, uvrsti klicajoči proces v vrsto čakajočih na dani semafor, sicer ga pusti naprej v izvajanje in popravi števec semaforjev. Princip čakanja je po sistemu FIFO /WIR76/.

2.3.3 IZMENJAVA SPOROČIL

Razporejevalnik poleg semaforjev omogoča tudi izmenjavo sporočil. čeprav lahko naredi izmenjavo sporočil uporabnik sam s semaforji, smo se odločili, da jo izvedemo še posebej, tako da zmanjšamo dodatno delo na najmanjšo možno mero.
Pošiljanje sporočil je izvedeno tako, da proces, ki pošilja sporočilo, ne preneha z izvajanjem tudi kadar ne more oddati sporočila. če sporočila ni mogoče oddati, dobi o tem ustrezen odgovor, vendar se njegovo stanje (TEKOČI) zaradi tega ne spremeni.
če je proces že dobil kako sporočilo, pa ga še vedno ni prevzel, so vsa nadaljnja pošiljanja temu procesu neveljavna vse dokler ga ne prevzame. Na ta način preprečimo, da bi se sporočila kopičila in porabila ves razpoložljivi prostor. Osnovna izvedba razporejevalnika omogoča pošiljanje naslova. Ker mora biti prostor za sporočilo izven prostora pošiljatelja (podiranje sklada), je prostor za sporočilo v jedru razporejevalnika.

2.4 SPEČI PROCESI

Včasih je potrebno, da proces sinhrono odstopi procesor za točno določen časovni interval. V ta namen je na razpolago poseben klic, ki postavi proces v vrsto spečih procesov. Za take procese je v razporejevalniku izdelana posebna struktura, ki procese razvrsti po naraščajočem času čakanja. časi čakanja so podani relativno glede na predhodnika. Tako razporejevalnik ob vsaki urini prekinitvi pogleda le prvi proces v vrsti in ga prebudi, če je njegov čas spanja že potekel.

2.5 STRATEGIJA RAZPOREJANJA

V osnovni izvedbi razporejevalnika je strategija razporejanja zelo preprosta. Procesi se med sabo lahko razlikujejo po prioriteti. Izvaja se le en proces in to proces z najvišjo prioriteto. Kadar je teh procesov več, se med seboj izmenjujejo in si enakovredno delijo procesorski čas. Procesi z nižjo prioriteto čakajo na izvajanje vse dokler vsi procesi z višjo prioriteto ne preidejo v eno izmed stanj skupine NEIZVRŠLJIV ali ne zapustijo sistema (klic KILL). Vsak proces ima pravico do neprekinjene uporabe procesorja N urinih prekinitev (če sam pred tem ne kliče kakega od podprogramov, ki mu spremeni stanje ali pa se ne zbudi kak proces z višjo prioriteto). Tak algoritem razporejanja se je izkazal kot zadovoljiv pri aplikacijah v realnem času, pri interaktivnem delu pa se ni izkazal ravno najbolje. Zato smo algoritem spremenili z uvedbo dinamične prioritete, tako da je podoben razporejevalniku na VAX VMS /VMS82/. Po končanem čakanju na zmogljivosti razporejevalnik dvigne prioriteto interaktivnega programa za določeno vrednost (to vrednost določimo z uglaševanjem). Ob vsakem poskusu prerazporeditve procesov se prioriteta tega procesa spusti, dokler ne pride do svoje začetne (osnovne) prioritete. Računski procesi (compute bound) so ves čas na isti (osnovni) prioriteti.

3. UPORABA

Razporejevalnik smo temeljito testirali in tudi uporabili v realnih aplikacijah. Izkazalo se je, da je razporejevalnik dovolj robusten in univerzalen za uporabo na različnih področjih.

3.1 PROBLEM LAČNIH FILOZOFOV

Problem lačnih filozofov je prvič predstavil Hoare /HOA85/. Gre za omejeno število zmogljivosti. Rešitev, ki je predstavljena v dodatku B, preprečuje smrtni objem, vendar ne preprečuje stradanja. Kot je pokazalo testiranje, do stradanja ni prišlo, če smo opazovali sistem dovolj dolgo. Predstavljena rešitev uporablja monitor za zaščito kritičnega področja.

3.2 DELOVNE POSTAJE

Razporejevalnik smo preizkusili tudi v realni aplikaciji. Gre za računalnik združljiv z IBM-PC, na katerega je priključenih več delovnih postaj, ki asinhrono pošiljajo podatke po komunikacijskem kanalu računalniku, ki jih mora urediti, zapisati na disk in narediti obdelavo zbranih podatkov ter jo izpisati na zaslon. Aktivnosti posameznih postaj so med seboj neodvisne. Izkazalo se je, da je razporejevalnik dobro orodje, ki omogoča na preprost in pregleden način programiranje aplikacij za paralelno procesiranje.

4. NADALJNJE DELO

Razporejevalnik bomo preizkusili še na nekaterih problemih procesiranja v realnem času. Poiskali bomo najboljšo dolžino časovnega intervala neprekinjenega izvajanja procesa v okolju, ki zahteva izjemno hiter odziv in obenem hitro obdelavo podatkov. Poleg tega načrtujemo tudi primerjalno analizo različnih strategij razporejanja procesorskega časa (scheduling policy).

Razen tega nameravamo razširiti sistem z modulom, ki bo uvedel dodatne konstrukte za lažje paralelno programiranje (COBEGIN in COEND, REGION... kot na primer CC-Modula /COL87/ in jih priporoča /BSI87/) in nadaljevati delo s paralelnimi algoritmi.

5. ZAKLJUČEK

Predstavljeni sistem se je izkazal kot dobro programsko orodje za delo v večopravilnem okolju tako na področju poskusov s paralelnimi algoritmi kot na področju aplikacij za končne uporabnike. Sistem je dovolj robusten in kompakten tudi za težke pogoje dela v industriji.

LITERATURA

/HAN73/ Hansen,P.B.: Operating Systems Principles, Prentice Hall, 1973

/DIJ75/ Dijkstra,E.W.: Guarded Commands, Nondeterminancy and Formal Derivation of Programms, Comm.ACM, v.18 n.8, 1975

/WIR76/ Wirth N.: Algorithms + Data Structure = Programs, Prentice Hall, 1976

/HOL78/ Holt R.C., G.S.Graham, E.D.Lasowska, M.A.Scott : Structured Concurrent Programming with Operating Systems Applications; Addison Wessley, 1978

/VMS82/ Digital Equipment Corporation: VAX Software Handbook, DEC, 1982

/FIL84/ Filman,R.E., Friedman D.P.: Coordinated computing: Tools and Techniques for Distributed Software, McGraw Hill, 1984

/COM84/ Comer,Douglas: Operating System Design, the Xinu Approach; Prentice Hall, 1984

/WIR85/ Wirth,N.:Programming in Modula-2, Springer Verlag, 1985

/HOA85/ Hoare,C.A.R.: Communicating Sequential Processes, Prentice Hall, 1985

/LOG86/ Logitech: Modula-2/86, Logitech, 1986

/JON87/ Jones,G.: Programming in Occam, Prentice Hall, 1987

/BSI87/ BSI Modula-2 Working Group: Standard Concurrent Programming Facilities, N 116 Issue 3, 1987

/BRO87/ Brodnik A.: Programiranje z modulo-2, Informatica, 1987

/COL87/ Collado M.: A Modula-2 Implementation Of CSP, ACM Sigplan Notices, Vol. 22, N6, June 1987

DODATEK A (DEFINITON MODULE PROCESS)

```
DEFINITION MODULE PROCESS;

  FROM SYSTEM IMPORT ADDRESS;

  EXPORT QUALIFIED
    tProcState,     (* tip moznih stanj procesa *)
    PCREATE, PRESUME, PKILL, PSUSPEND, PSETPRIO, PGETPID,
    PGETSTAT, PGETCURR, PSLEEP, PRECEIVE, PSEND,
    tPid, tPrio, tSemaphore,
    cProcNameLen,
    tProcName, tProcStatus, tStatBlock,
    SCREATE, SINIT, SDELETE, SWAIT, SSIGNAL, SSTATUS, SGETSID,
    cSemNameLen,
    tSemName, tSemStatus,
    cPrio1,  cPrio2, cPrio3, cPrio4,  cPrio5, cPrio6, cPrio7;

  CONST
    cSemNameLen = 20;
    cProcNameLen = 20;          (* Process name lenghth *)

  TYPE
    tProcState = (PrCurr,       (* Process Current *)
                  PrReady,      (* Process Ready *)
                  PrSusp,       (* Process Suspended *)
                  PrWait,       (* Process Wait *)
                  PrSleep,      (* Process Sleeping *)
                  PrRec,        (* Process Receiving *)
                  PrFree        (* Process Slot in Table not Used *) );

    tProcName = ARRAY [0..cProcNameLen-1] OF CHAR;
    tProcStatus = (NOTOK,OK);
    tStatBlock = RECORD   (* undeveloped *)
                   status : CARDINAL
                 END;
    tPrio;                      (* process priority - type *)
    tPid;                       (* process - type *)
    tSemaphore;                 (* semaphore - type *)
    tSemName = ARRAY [0..cSemNameLen-1] OF CHAR;
    tSemStatus = (SNOTOK, SOK);

  VAR
    cPrio1, cPrio2, cPrio3, cPrio4, cPrio5, cPrio6, cPrio7 : tPrio;

  PROCEDURE PCREATE (process:PROC; envsize:CARDINAL; name:ARRAY OF CHAR;
                     VAR pid : tPid; prio: tPrio) : tProcStatus;
    (* Creates a new process with PrSusp status *)

  PROCEDURE PRESUME  (pid : tPid) : tProcStatus;
    (* Resumes a suspended process (PrSusp --> PrReady) *)

  PROCEDURE PKILL    (pid : tPid) : tProcStatus;
    (* Deletes a process and releases its resources (PrFree) *)

  PROCEDURE PSUSPEND (pid : tPid) : tProcStatus;
    (* Suspends a process (PrReady or PrCurr --> PrSusp) *)

  PROCEDURE PSETPRIO (pid : tPid; prio : tPrio) : tProcStatus;
    (* Change process priority *)

  PROCEDURE PGETPID  (name:ARRAY OF CHAR; VAR pid : tPid) : tProcStatus;
    (* returns process id (Pid) of process NAME *)

  PROCEDURE PGETSTAT (pid : tPid; VAR statblock:tStatBlock) : tProcStatus;
    (* returns process status informaton *)

  PROCEDURE PGETCURR () : tPid;
    (* returns process own process id *)

  PROCEDURE PSLEEP (time : CARDINAL):tProcStatus;
    (* Sleep for time ticks (PrCurr --> PrSleep) *)

  PROCEDURE PRECEIVE (VAR what:ADDRESS; VAR who:tPid): tProcStatus;
    (*  Receive  a message from anybody (conditional  PrCurr  --> PrRec) *)

  PROCEDURE PSEND (whom:tPid; what:ADDRESS) : tProcStatus;
    (* Send a message to a process *)

  PROCEDURE SCREATE (name:tSemName; VAR sem:tSemaphore; count:CARDINAL):
                     tSemStatus;
    (* Create a new sempahore and return its ID in sem *)
```

```
  PROCEDURE SINIT (sem:tSemaphore; count:CARDINAL) : tSemStatus;
    (* Init a Semaphore to initial value count *)

  PROCEDURE SDELETE (sem:tSemaphore) : tSemStatus;
    (* Delete a semaphore *)

  PROCEDURE SWAIT  (sem:tSemaphore) : tSemStatus;
    (* Wait for a semaphore (PrCurr --> PrWait) *)

  PROCEDURE SSIGNAL (sem:tSemaphore) : tSemStatus;
    (* Signal on semaphore sem *)

  PROCEDURE SSTATUS  (sem:tSemaphore) : tSemStatus;
    (* Get Semaphore status information *)

  PROCEDURE SGETSID (name:tSemName;VAR sem:tSemaphore;count:CARDINAL)
                    : tSemStatus;
    (* Get the ID of semaphore name *)

END PROCESS.
```

DODATEK B (DINING PHILOSOPHERS)

```
MODULE SPAGHETTI;
FROM PROCESS IMPORT PCREATE, PRESUME, PSLEEP, PSUSPEND, PGETCURR,
                    PRECEIVE, PSEND, PKILL, PSETPRIO,
                    tPid, tProcStatus,
                    cPrio1, cPrio2, Port,
                    SCREATE,  SWAIT, SSIGNAL, SDELETE,
                    tSemName, tSemStatus, tSemaphore,
                    CURSOR, WRITEI, WRITE;
FROM InOut IMPORT WriteString, WriteCard, WriteLn, WriteHex;
FROM Break IMPORT EnableBreak;
FROM Strings IMPORT Concat;
FROM SYSTEM IMPORT GETREG, ADDRESS, WORD, BYTE, INBYTE, OUTBYTE, CODE;
FROM Keyboard IMPORT KeyPressed, Read;
FROM Devices IMPORT  SaveInterruptVector, RestoreInterruptVector;
FROM MyRandom IMPORT Random;


TYPE
  tPhil = [0..4];

VAR
  pid1,pid2,pid3,pid4,pid5 : tPid;
  procstat, status: tProcStatus;
  chn,I,J: INTEGER;
  w: WORD;
  konec : BOOLEAN;
  ch: CHAR;
  vec: ADDRESS;

MODULE FORKS [7];
IMPORT tSemaphore, tPhil, tProcStatus, tSemStatus, SWAIT, SCREATE, SSIGNAL,
       tSemName, Concat, PSLEEP;
EXPORT PICKUP, PUTDOWN;
  TYPE
      tNumOfFork = [0..2];

  VAR
      numOfForks : ARRAY tPhil OF tNumOfFork;
      ready : ARRAY tPhil OF tSemaphore;
      semstat : tSemStatus;
      i : INTEGER;
      tmp : tSemName;

PROCEDURE PICKUP (phil : tPhil);
  VAR
    left, right : tPhil;
BEGIN
  IF (numOfForks[phil] < 2) THEN
    semstat := SWAIT (ready[phil])
  END;
  right := phil;
  left  := (phil + 1) MOD 5;
  DEC(numOfForks[left]);
  DEC(numOfForks[right]);
END PICKUP;
```

```
PROCEDURE PUTDOWN (phil : tPhil);
  VAR
      left, right : tPhil;
BEGIN
  right := phil;
  left  := (phil + 1) MOD 5;
  INC(numOfForks[left]);
  INC(numOfForks[right]);
  IF numOfForks[left] = 2 THEN
    semstat := SSIGNAL (ready[left]);
  END;
  IF numOfForks[right] = 2 THEN
    semstat := SSIGNAL (ready[right]);
  END;
END PUTDOWN;

BEGIN  (* FORKS *)
  FOR i:= 0 TO 4 DO
    CASE i OF
      0 : tmp := 'Bacon' |
      1 : tmp := 'Sokrates' |
      2 : tmp := 'Aristoteles' |
      3 : tmp := 'Nietsche' |
      4 : tmp := 'Hegel'
    END (* case *);
    semstat := SCREATE (tmp, ready[i], 0);
    numOfForks[i] := 2;            (* forks initialization *)
  END;
END FORKS;

PROCEDURE First;
VAR
    timesEaten : CARDINAL;
BEGIN
  timesEaten := 0;
  LOOP
    PICKUP(0);
    WriteString ('Bacon eating ');
    WriteCard(timesEaten, 5);
    WriteLn;
    INC(timesEaten);
    PUTDOWN(0);
    procstat := PSLEEP (Random(10) );
  END;
END First;

PROCEDURE Second;
VAR
    timesEaten : CARDINAL;
BEGIN
  timesEaten := 1;
  LOOP
    PICKUP(1);
    WriteString ('Sokrates eating ');
    WriteCard(timesEaten, 5);
    WriteLn;
    INC(timesEaten);
    PUTDOWN(1);
    procstat := PSLEEP (Random(10) );
  END;
END Second;

PROCEDURE Third;
VAR
    timesEaten : CARDINAL;
BEGIN
  timesEaten := 1;
  LOOP
    PICKUP(2);
    WriteString ('Aristoteles eating ');
    WriteCard(timesEaten, 5);
    WriteLn;
    INC(timesEaten);
    PUTDOWN(2);
    procstat := PSLEEP (Random(10) );
  END;
END Third;
```

```
PROCEDURE Fourth;
VAR
    timesEaten : CARDINAL;
BEGIN
  timesEaten := 1;
  LOOP
    PICKUP(3);
    WriteString ('Nietsche eating ');
    WriteCard(timesEaten, 5);
    WriteLn;
    INC(timesEaten);
    PUTDOWN(3);
    procstat := PSLEEP (Random(10) );
  END;
END Fourth;

PROCEDURE Fifth;
VAR
    timesEaten : CARDINAL;
BEGIN
  timesEaten := 1;
  LOOP
    PICKUP(4);
    WriteString ('Hegel eating ');
    WriteCard(timesEaten, 5);
    WriteLn;
    INC(timesEaten);
    PUTDOWN(4);
    procstat := PSLEEP (Random(10) );
  END;
END Fifth;


BEGIN
  EnableBreak;
  status := PSETPRIO(PGETCURR(),cPrio2);
  status := PCREATE(First,800H,'Bacon',pid1,cPrio1);
  status := PCREATE(Second,800H,'Sokrates',pid2,cPrio1);
  status := PCREATE(Third,800H,'Aristoteles',pid3,cPrio1);
  status := PCREATE(Fourth,800H,'Nietsche',pid4,cPrio1);
  status := PCREATE(Fifth,800H,'Hegel',pid5,cPrio1);
  status := PRESUME(pid1);
  status := PRESUME(pid2);
  status := PRESUME(pid3);
  status := PRESUME(pid4);
  status := PRESUME(pid5);
  konec := FALSE;
  LOOP
    Port;
    status := PSLEEP(18);
    IF KeyPressed() THEN
      EXIT;
    END;
  END;  (* LOOP *)
  status := PKILL(PGETCURR());
  WriteString('END - Philosophers');
END SPAGHETTI.
```

TERMINOLOGIJA PROGRAMIRANJA V PROLOGU
TERMINOLOGY OF PROLOG PROGRAMMING

INFORMATICA 2/88

Igor Kononenko
Fakulteta za elektrotehniko

Nada Lavrač
Inštitut Jožef Stefan

UDK 681.3.06. Prolog: 001.4

Prolog je novejši programski jezik, ki se vse bolj uveljavlja tudi pri nas. Zaradi velikih razlik v terminologiji tako v svetu kot pri nas je potrebno poenotiti terminologijo, ki se uporablja v logičnem programiranju ter programiranju v prologu. Osnova za delo sta slovarja v (Kononenko 87) in (Kononenko, Lavrač 87). V prispevku smo podali razlike med terminologijama matematične logike in logičnega programiranja ter med prologom in ostalimi programskimi jeziki. Poskusili smo odpraviti tudi nekatere terminološke nesporazume. Podani so tudi prevodi nekaterih izrazov iz angleščine v slovenščino, za katere smo se zedinili na Fakulteti za elektrotehniko in Institutu Jožef Stefan v Ljubljani.

ABSTRACT

Lately, Prolog became a very popular language in the computer programming community. As there is a lot of diversity in the terminology we point out some of the main problems and try to eliminate them by defining the most useful terms. The basis for this paper is a glossary in (Kononenko & Lavrač 87). We present differences between mathematical logic and logic programming, between Prolog and other programming languages and show how these differences affect the terminology. We also point out some misunderstandings regarding the usual terminology and mention some synonyms used in the field of logic programming.

1. RAZLIKE MED MATEMATIČNO LOGIKO IN LOGIČNIM PROGRAMIRANJEM

Prolog (Clocksin, Mellish 81) je najbolj razširjen jezik logičnega programiranja. Ime "PROLOG" je dejansko kratica za 'PROgramming in LOGic'. Ker prolog temelji na matematični logiki, je iz nje privzeta tudi terminologija, kar je v določenih primerih pripeljalo do dvoumnosti.

1. MATHEMATICAL LOGIC AND LOGIC PROGRAMMING

Prolog (Clocksin & Mellish 81) is the most widespread language of the logic programming. PROLOG is in fact an abbreviation for PROgramming in LOGic. As it is based on mathematical logic the similar terminology is adopted which in some cases leads to ambiguities.

Osnovna razlika med stavki v logiki in stavki v prologu je ta, da imajo stavki v prologu tudi proceduralen in ne le deklarativen pomen. Pogosto se uporablja izraz 'predikat' kot sinonim za 'proceduro'. Procedura je del programa (množica stavkov), ki definira predikat z določenim številom argumentov. Definicija je kljub deklarativnemu značaju proceduralna, zato je izraz 'predikat' zavajajoč. V logičnem programiranju zadostuje uporaba izrazov 'procedura', 'vgrajena procedura' ipd. namesto izrazov 'predikat', 'vgrajeni predikat', ipd.

Pogosto se v literaturi mešata izraza 'prilagajanje' in 'unifikacija'. Prilagajanje v prologu je proceduralna implementacija unifikacije iz matematične logike, ki ponavadi zaradi učinkovitosti deluje drugače kot unifikacija. Neopredeljena spremenljivka se lahko prilagodi z izrazom, v katerem sama nastopa, ne more pa se z njim unificirati.

Preglavice povzročajo tudi izrazi 'atom', 'literal' in 'cilj'. Atom je v matematični logiki kratica za atomarno formulo, medtem ko v prologu pomeni vrsto preprostih nedeljivih podatkovnih konstruktov, ki so konstante različne od števil. Tako ustreza atom v prologu kostanti v matematični logiki. Atom v matematični logiki ustreza strukturi v prologu, ki predstavlja klic procedure, če se nahaja v telesu stavka (v pogojnem delu stavka), ali pa sklep, če se nahaja v glavi stavka (v sklepnem delu stavka).

Klicu procedure v telesu prologovega stavka pravimo cilj. Nekateri avtorji (npr. Sterling, Shapiro 86) pravijo glavi stavka tudi cilj. Sintaktično so glava stavka in cilji v telesu stavka pozitivni (nenegirani) literali. Literal je sintaktični konstrukt sestavljen iz imena predikata in seznama argumentov v oklepaju, ločenih z vejicami.

Funktor v matematični logiki določa funkcijo, medtem ko v prologu funkcij ni. Funktor v prologu omogoča konstrukcijo struktur z imenom in mestnostjo (številom argumentov) danega funktorja. Strukture so podatkovni objekti v prologu in nimajo veliko skupnega s funkcijami v matematiki.

Nerodno je, da sta se pri programiranju v

The main difference between logical statements and Prolog clauses is that Prolog clauses have also the procedural meaning while logical statements have only the declarative meaning. The term 'predicate' is often used as a synonym for a 'procedure'. A procedure is a part of a program (a sequence of clauses) that defines a predicate with the given name and arity. Although the definition may be viewed declaratively it is in fact procedural (the order of clauses and the order of goals in clauses are important). The term 'predicate' is therefore misleading. For programming purposes it is thus more convenient to use terms 'procedure', 'built-in procedure', etc. instead of 'predicate', 'built-in predicate', etc.

In the literature is often made no explicit distinction between 'unification' and 'matching'. Matching in Prolog is a procedural implementation of the unification from mathematical logic and usualy differs from it (for the efficiency reasons). It allows an uninstantiated variable to be matched with a term in which it itself appears while it cannot be unified with such a term (this problem is usually referred to as the 'occurs check' problem).

There are difficulties with terms 'atom', 'literal' and 'goal'. In mathematical logic an atom is an abbreviation for 'atomic formula' while in Prolog an atom is a simple data structure, i.e. a constant that is not a number. Therefore an atom in Prolog corresponds to a constant in mathematical logic. An atom in mathematical logic corresponds to a Prolog structure representing a procedure call if it appears in a body of a clause (a condition part) or a conclusion if it appears in the head of a clause (a conclusion part).

A procedure call in a body of a clause is usually called a goal. Some authors (e.g. Sterling & Shapiro 86) call the head of a clause also a goal. Syntactically the head of a clause and goals in the body of a clause are positive (nonnegated) literals. A literal is constructed from a predicate name and a list of arguments enclosed in brackets and separated by commas. A positive literal is a synonym for atomic formula. It is better to use a term 'literal' because of the previously mentioned definition of an atom in Prolog.

prologu v angleščini udomačila izraza 'bound variable' in 'free variable' za spremenljivko, ki ima oziroma nima vrednosti. V matematični logiki se namreč ta dva izraza uporabljata za kvantificirano (vezano) in nekvantificirano (nevezano) spremenljivko. Zato je bolje, da se namesto 'bound' in 'free' uporabljata izraza 'instantiated' in 'uninstantiated'. Možni prevodi so instancirana (neinstancirana), prilagojena (neprilagojena) ali opredeljena (neopredeljena). Zadnji prevod je najustreznejši, saj se vrednost spremenljivke v prologu lahko med izvajanjem bolj ali manj opredeli.

2. RAZLIKE MED PROLOGOM IN OSTALIMI PROGRAMSKIMI JEZIKI

Za razliko od ostalih razširjenih programskih jezikov, ki imajo proceduralni značaj, je prolog je deklarativni jezik. Proceduralni pomen programa v prologu je definiran z načinom izvajanja prologovega interpreterja. Vsekakor je v angleščini pravilneje uporabljati izraz 'execution' za izvajanje prologovega programa namesto ustaljenega izraza 'computation' (računanje), saj je izvajanje prologovega programa dejansko sklepanje (inference) na osnovi pravil in dejstev. Zato računalnikom vse pogosteje pravijo tudi 'inference machine' (stroj za sklepanje) namesto 'computer'.

Med izvajanjem prologovega programa spremenljivke dobijo svojo vrednost s procesom prilagajanja. Pravimo, da vrednost spremenljivke postane (bolj) določena oziroma opredeljena. Ne more pa se vrednost spremenljivke spremeniti na neko povsem drugo vrednost, razen pri avtomatskem vračanju. Pri avtomatskem vračanju vrednost lahko postane manj opredeljena. Za razliko od prologa pa se vrednosti spremenljivk v proceduralnih jezikih lahko spreminjajo. Zato ima izraz 'vrednost spremenljivke' v prologu nekoliko drugačen pomen.

Prireditev v standardnih proceduralnih jezikih pomeni spremembo vrednosti spremenljivke na neko določeno vrednost. V prologu take prireditve ni. Izraz 'prireditev' se uporablja v prologu za vgrajeno proceduro 'is', ki se uporablja za izračun vrednosti aritmetičnega izraza, in prilagoditev izračunane vrednosti s spremenljivko ali konstanto, če je to možno.

In mathematical logic a functor determines a function while in Prolog are no functions. Functors in Prolog are used to construct compound data structures with the given name and arity. Structures in Prolog have not much in common with mathematical functions.

It is awkward that the term 'bound variable' is used for a variable taht has a value and 'free variable' for a variable that doesn't have a value. In mathematical logic bound variable represents a quantified and free variable represents an unquantified variable. In Prolog all variables are universally quantified (except in questions). Therefore it is better to use terms 'instantiated' and 'uninstantiated variable'.

2. PROLOG AND OTHER PROGRAMMING LANGUAGES

Prolog is a declarative language and it largely differs from other popular programming languages which are procedural. The procedural meaning of a Prolog program is defined with the way how it is executed by the Prolog interpreter. It is better to use the term 'execution' than 'computation'. Execution of a Prolog program can be viewed as inference and a computer can be also called an inference machine.

During the execution of a Prolog program variables get their values by matching. We say that the value of a variable becomes (more) specified or determined. While backtracking the value may become less specified. A variable is therefore more or less instantiated. The value of a variable cannot be changed as in other programming languages. Thus in Prolog the term 'the value of a variable' has a slightly different interpretation than in other programming languages.

An assignment in standard procedural languages causes the change of the value of a variable to a certain new value. There is no such assignment in Prolog. The term 'assignment' is used in Prolog for the built-in procedure 'is' which computes the value of the arithmetical expression on the right-hand side and matches the result with the argument on the left-hand side, if this is possible.

Pri programiranju niz (string) predstavlja ponavadi niz znakov med dvema enojnima narekovajima zgoraj. V prologu je niz seznam celih števil, ki ustrezajo ASCII kodam znakov, in ga lahko namesto v standardni notaciji seznama napišemo tudi kot niz ustreznih znakov med dvojnima narekovajima zgoraj. Niz znakov med enojnima narekovajima zgoraj pa v prologu predstavlja atom.

3. DVOUMNOSTI V TERMINOLOGIJI

Prioriteta operatorjev je v prologu definirana z celimi števili (ponavadi od 1 do 1200). Čim manjše je število, tem močneje veže operator in obratno. Ta nekoliko neobičajna definicija dostikrat pripelje do nerazumevanja.

V literaturi se pogosto navajata 'green cut' (zeleni rez) in 'red cut' (rdeči rez) (glej npr. Bratko 86). Brez zelenega reza bi procedura pravilno delovala, le nekoliko počasnejše bi bilo izvajanje zaradi nepotrebnega vračanja. Rdečega reza pa iz procedure ne smemo odstraniti, ker bi bilo izvajanje napačno. Mnogi avtorji napačno ugotavljajo, da zeleni rez ne spremeni deklarativnega pomena medtem ko rdeči rez spremeni deklarativni pomen procedure.

Rez je kontrolni konstrukt, ki nima deklarativnega pomena in zato tudi ne more spremeniti deklarativnega pomena. Vpliva pa na proceduralni pomen dane procedure, saj se zaporedje izvajanja z dodajanjem reza spremeni. Torej tako zeleni kot rdeči rez spremenita proceduralni pomen procedure, le da zeleni rez lahko odstranimo, ne da bi spremenili proceduralno pravilnost procedure, medtem ko z odstranitvijo rdečega reza postane procedura nepravilna.

4. SINONIMI

V seznamu sinonimov so ustreznejši izrazi na desni strani:

atomarna formula - pozitivni literal

objekt - izraz

podatkovna struktura - izraz

In programming the term 'string' usually represents a string of characters enclosed in single quotes. In Prolog a string is represented with a string of characters enclosed in double quotes and is equivalent to a list of integers that correspond to ASCII codes of characters in the string. In Prolog a string of characters enclosed in single quotes is an atom.

3. AMBIGUITIES IN TERMINOLOGY

The precedence of an operator is defined as a strength with which an operator binds its arguments. It is labeled with an integer (usually between 1 and 1200). The greater the label is the weaker the operator binds its arguments and vice versa. This somehow unusual definition often leads to misunderstandings.

In the literature we often meet 'green cut' and 'red cut' (e.g. Bratko 86). Without green cut the procedure would still perform correctly although less efficiently due to unnecessary backtracking. The red cut must not be removed because the execution of the procedure would be incorrect. Many authors incorrectly state that the green cut does not affect the declarative meaning while the red cut does. Cut is a procedural construct with no declarative meaning and thus cannot affect the declarative meaning. It certanly affects the procedural meaning of a procedure because the execution is changed. Therefore both green and red cuts affect the procedural meaning of a procedure. The green cut can be removed without affecting the procedural correctness of the procedure while when the red cut is removed the procedure becomes incorrect.

4. SYNONYMS

In the following list of synonyms the terms at the right-hand side should be preferred.

atomic formula - (positive) literal

bound variable - instantiated variable

predefinirana procedura - vgrajena procedura

sistemska procedura - vgrajena procedura

struktura - sestavljeni izraz

term - izraz

vgrajeni predikat - vgrajena procedura

5. USTALJENI PREVODI NEKATERIH IZRAZOV

Za naslednje prevode izrazov smo se zedinili na Fakulteti za elektrotehniko in Inštitutu Jožef Stefan v Ljubljani:

cut - rez

instance - primer

instance of a term - primer izraza

instantiated - opredeljen

matching - prilagajanje

parent goal - nadrejeni cilj

term - izraz

ZAHVALA

Slovarja v (Kononenko 87) in (Kononenko, Lavrač 87), ki sta osnova tega dela, sta rezultat skupnega prizadevanja ob pomoči mnogih sodelavcev. Zahvaljujeva se Tatjani Janc in Alenu Varšku za pripombe na rokopis.

built-in predicate - built-in procedure

compound term - structure

computation - execution

control predicate - control procedure

data structure - term

evaluable predicate - built-in procedure

free variable - uninstantiated variable

joint variable - shared variable

object - term

predefined operator - built-in operator

predefined procedure - built-in procedure

priority - precedence

system procedure - built-in procedure

stopping condition - boundary condition

unbound variable - uninstantiated variable

AKNOWLEDGEMENTS

Many colleagues helped us in the preparation of a glossary in (Kononenko & Lavrač 87) which is the basis for this paper. We thank Tatjana Janc and Alen Varšek for their remarks on a manuscript.

LITERATURA - REFERENCES

Bratko, I. (1986) Prolog programming for artificial intelligence, Addison-Wesley.

Clocksin, W.F., Mellish, F.G. (1981) Programming in prolog, Springer-Verlag.

Kononenko, I. (1987) Uvod v prolog in zbirka nalog z rešitvami, skripta, Fakulteta za elektrotehniko, Ljubljana.

Kononenko, I., Lavrač, N. (1987) Prolog through examples - A practical programming guide, Sigma Press.

Sterling, L., Shapiro, E. (1986) The art of prolog - Advanced programming techniques, MIT Press.

UDK 681.3.02

Jože Rugelj
Institut »Jožef Stefan«

Sinhronizacijski mehanizmi v porazdeljenih računalniških sistemih so potrebni za definicijo in realizacijo urejenosti dogodkov v računalniškem sistemu. Članek podaja pregled mehanizmov za sinhronizacijo na različnih nivojih abstrakcije računalniškega sistema.

Synchronization mechanisms in distributed computing systems are necessary for the definition and the realization of event ordering in the computing system. This article gives an overview of mechanisms, used on different levels of abstraction.

1. UVOD

Porazdeljen računalniški sistem je množica procesnih elementov. Vsak procesni element ima lasten pomnilnik in procesne zmogljivosti. V procesnih elementih se izvajajo procesi. Z oznako porazdeljenost je mišljena porazdelitev nadzora in ne nujno tudi prostorska porazdeljenost. V tesno sklopljenih sistemih so procesi povezani med seboj s skupnim pomnilnikom, v šibko sklopljenih sistemih pa so procesna mesta povezana s komunikacijskimi kanali, ki skupaj tvorijo komunikacijsko mrežo. Skupnega pomnilnika v takih sistemih ni. V tem primeru gre tudi za prostorsko porazdelitev procesnih elementov, ki jim zaradi tega rečemo tudi procesna mesta.
Procesne elemente združuje v porazdeljen sistem porazdeljen operacijski sistem. Glavne naloge porazdeljenega operacijskega sistema so podpora medprocesne komunikacije, dodeljevnje virov in upravljanje z njimi, upravljanje z imeni ter reševanje iz napak. Jedra porazdeljenega operacijskega sistema na posameznih mestih so lahko implementirana kot jedra osnovnega operacijskega sistema na tem mestu ali kot procesi na uporabniškem nivoju /Trip87/.
V sistemih z večimi procesnimi elementi se procesi izvajajo sočasno, dokler ne potrebujejo medsebojnih stikov. Načrtovane in vodene stike med procesi imenujemo procesna komunikacija in sinhronizacija. Procesi morajo sodelovati zaradi omejevanja sočasnosti in zaradi medsebojnega razvrščanja /Verj83/. Odnose med procesi bi glede na način sodelovanja lahko razdelili v dve glavni kategoriji: lahko tekmujejo med seboj ali so v odnosu proizvajalec-potrošnik /Hwan85/.

2. KONSISTENTNOST IN NEDELJIVOST

Predpostavljamo, da se procesi izvajajo v diskretnih korakih in na vsakem koraku generirajo dogodek. Dogodek je lahko lokalen procesu in ga drugi procesi ne opazijo ali pa je viden vsem in vsebuje problem sinhronizacije.

Na vsakem nivoju abstrakcije lahko proces, ki se izvaja na nekem procesnem mestu, sproži operacije. Operacija na nivoju j je implementirana kot množica aktivnosti na nivoju i, pri čemer velja i < j. Kar vidimo z nivoja j kot aktivnost je definirano na nivoju i kot operacija. Tak model velja rekurzivno za vse nivoje.

Operacije oziroma aktivnosti vodijo in upravljajo vire. Viri so predstavljeni kot podatkovni objekti. Na primer, periferna naprava je lahko predstavljena s svojim trenutnim stanjem, ki je določeno z vrednostmi množice parametrov. Zapis na datoteki lahko predstavimo z njegovo vsebino. Primeri aktivnosti, ki delajo s podatkovnimi objekti, so pisanje, branje, kreiranje in brisanje.
Podatkovni objekti imajo med seboj semantične povezave, to pomeni, da morajo njihove vrednosti zadoščati nekim omejitvam. Ko objekt kreiramo, brišemo ali vanj pišemo, je pogosto potrebno kreirati, brisati ali pisati še v druge objekte, da bi zadostili konsistenčnim omejitvam /LeLa81/.

Vsaka operacija je definirana tako, da predpostavlja na vhodu množico objektov s konsistentnimi vrednostmi in zagotavlja kot rezultat množico novih vrednosti, ki so tudi konsistentne.
Torej, če so konsistentna stanja, ki se prenašajo med procesi ali so shranjena, je tudi procesiranje konsistentno.

Če ni posebnih predpostavk, lahko ohranimo konsistentnost samo z zagotavljanjem nedeljivosti operacij. Z definiranjem nekaterih drugih zahtev lahko zahtevo za nedeljivost opustimo. Definirajmo nedeljivost operacije. Operacija je nedeljiva, če zadošča naslednjima pogojema:

1. ali se izvedejo vse aktivnosti popolnoma ali pa se ne izvede nobena in

2. vmesna stanja pri izvajanju operacije

niso vidna nobeni drugi operaciji.

Nedeljivost je dobro poznan koncept /Lmps81/.

Običajne instrukcije na računalniku so nedeljive, ker se izvajajo strogo zaporedno na eni procesni enoti. V porazdeljenih sistemih se aktivnosti, ki pripadajo dani operaciji, lahko izvajajo na različnih procesnih enotah brez določenih časovnih povezav. Zato implementacija nedeljivih operacij zahteva specifične mehanizme, ki jih v centraliziranih sistemih ne potrebujemo.

Začetek izvajanja operacije šele po izvršitvi prejšnje operacije vodi k posebni vrsti dodeljevanja imenovanega zaporedno dodeljevanje (serial scheduling). V porazdeljenih sistemih pa je izključna uporaba zaporednega dodeljevanja zelo neučinkovita. Če aktivnosti, ki jih sproži dana množica operacij, delujejo nad različnimi objekti, vzporedno izvajanje aktivnosti ni le možno, ampak celo priporočljivo. Še več, če so aktivnosti, ki delujejo nad danim objektom in pripadajo različnim operacijam, sprožene sočasno in postavljene v čakalno vrsto procesnega elementa, ki vsebuje objekt, je nelzkoriščen čas med dvema zaporednima aktinostima krajši kot v primeru, če je naenkrat sprožena samo ena aktivnost.

Torej je zaželjeno, da dovoljujemo prepletanje aktivnosti (interleaving), kolikor je to mogoče, in s tem optimiziramo zmogljivosti. Odgovarjajoča dodeljevanja imenujemo dodeljevanja s prepletanjem in nekatera od njih so ekvivalentna zaporednim, torej so konsistentna. Toda v splošnem to ne velja. Če torej želimo čim večjo paralelnost in s tem veliko hitrost in dobro izkoriščenost virov moramo poiskati ustrezno prepletanje, ki ohranja konsistentnost. S sinhronizacijskimi mehanizmi pa lahko realiziramo potrebne odnose med procesi oz. operacijami.

3. RAZVRŠČANJE DOGODKOV

Določanje, kateri dogodek se je zgodil prej, je običajno intultivno zasnovano na razmišljanju v fizikalnem časovnem svetu. Tak pristop predpostavlja, da lahko definiramo nek univerzalen čas, ki je dosegljiv z različnih lokacij v sistemu. Vemo pa, da taka rešitev ne obstaja. V praksi lahko dobimo približke univerzalnega časa z dano natančnostjo. Vendar pa ni potrebno ali pa ni mogoče izraziti sistemskih specifikacij s fizikalnim časom /Kope87/.

Obstaja določena kronološka urejenost dogodkov v sistemu, ker procesi, ki generirajo te dogodke, spoštujejo specifična pravila (algoritme, protokole), ki izražajo relativno urejenost v množici dogodkov opazovanega sistema.

Taka urejenost dovoljuje procesom, ki opazujejo dogodke, da pravilno implementirajo sistemske aktivnosti. Glede na naravo teh aktivnosti je potrebna delna ali popolna urejenost dogodkov.

Delno urejenost, označeno z " --> " (beri "se zgodi pred") lahko definiramo nad katerokoli množico dogodkov, ki jih generirajo procesi, ki izmenjujejo sporočila /Lamp78/.
Za relacijo "se zgodi pred" velja:

1. če sta a in b aktivnosti istega procesa in se a zgodi pred b, potem je a --> b
2. če je a aktivnost za pošiljanje sporočila iz enega procesa in je b aktivnost, ki sprejme to sporočilo v drugem procesu, velja a --> b
3. če je a --> b in b --> c, potem velja a --> c

Če procesi komunicirajo med seboj z izmenjavanjem sporočil, lahko razvrstitimo nekatere dogodke, ki se zgodijo v različnih procesih. Urejenost " --> " je samo delna. Npr. če imamo dogodka a in b in velja a --> b in b --> a, potem je nemogoče reči, kateri od dogodkov se je zgodil prej. Za take dogodke rečemo, da so sočasni (concurrent). V splošnem dogodkov iz procesov, ki med seboj ne komunicirajo, ne moremo razvrstiti. V odvisnosti od omejitev, ki jih moramo upoštevati, je to dopustno ali pa tudi ne. Eden od načinov za dosego popolne urejenosti, kadar je le-ta nujna, je pridobitev popolne urejenosti iz delne z definicijo popolne urejenosti nad množico procesov.

Omeniti moramo enega od osnovnih problemov računalniških sistemov, to je problem zakasnitev pri procesiranju. V enoprocesorskih sistemih lahko zelo natančno določimo čas, ki ga procesna enota potrebuje za izvršitev danega ukaza. Seveda pa v splošnem operacijski sistemi niso zasnovani na osnovi takih podatkov. Še manj pa bi bila taka zasnova upravičena za porazdeljene operacijske sisteme, saj zakasnitve vključujejo še čas, ki je potreben za prenos podatkov in ukazov med posameznimi procesnimi enotami po komunikacijski mreži. V porazdeljenih sistemih, kjer se operacije izvajajo na različnih procesnih enotah, spreminjanje zakasnitev v komunikacijskem podsistemu lahko zmoti določeno urejenost dogodkov, ki jo pričakujemo. To se lahko zgodi celo v sistemih, kjer so zakasnitve stalne. Zato moramo uporabljati določene sinhronizacijske mehanizme pri procesih, kjer se dogodki zgodijo in pri procesih, ki opazujejo te dogodke.

Namen sinhronizacijskih mehanizmov je definiranje in realizacija razvrstitve poljubne množice dogodkov. Natančneje, rekli bomo, da je sinhronizacija način za definicijo in realizacijo delne all popolne urejenosti nad neko množico dogodkov.

Sinhronizacijskih mehanizmi nudijo procesom pripomočke, s katerimi se sistem ohranja v konsistentnem stanju. Stanje računalniškega sistema je konsistentno, če v vsakem trenutku ustreza nekim zunanjim določilom.

4. SPLOŠNE ZNAČILNOSTI SINHRONIZACIJSKIH MEHANIZMOV

Sinhronizacijski mehanizmi temeljijo na opazovanju in spreminjanju določenih skupnih sinhronizacijskih spremenljivk. V tesno sklopljenih sistemih so le-te v skupnem pomnilniku, v šibko sklopljenih sistemih pa so na nekaterih all na vseh procesnih mestih. Zahteve za branje ali spreminjanje spremenljivk in njihove vrednosti se prenašejo kot sporočila po komunikacijski mreži.

Pri porazdeljenih rešitvah so sinhronizacijske spremenljivke podvojene, razcepljene ali porazdeljene med mesta v sistemu.
Podvojenost ali celo pomnoženost spremenljivke pomeni, da so verzije iste spremenljivke v vseh ali vsaj v večih mestih v sistemu. Ustrezen mehanizem poskrbi za delno ali popolno konsistentnost v vsakem trenutku.
Razcepljenost spremenljivk je razdelitev vrednosti spremenljivke na več komponent. Komponente so porazdeljene po mestih, kjer se spreminja njihova vrednost. Prava vrednost spremenljivke je linearna kombinacija vrednosti njenih komponent.
Porazdelitev spremenljivk pa je rešitev, pri kateri so sinhronizacijske spremenljivke porazdeljene po različnih mestih v sistemu, vendar samo po ena verzija vsake.

Za izvedbo operacij nad sinhronizacijskimi spremenljivkami na katerem koli nivoju abstrakcije se lahko sklicujemo na določen osebek, ki ga imenujemo centralni sinhronizacijski osebek. Le-ta je dostopen vsakemu proizvajalcu vsakič, ko začne novo operacijo.
Termin "sinhronizacijski osebek" uporabljamo za usklajevalce procesov , semaforje, monitorje ipd.; to so torej aktivnosti, ki popravljajo in spremljajo stanje sinhronizacijskih spremenljivk.

Sinhronizacijski osebek je centralen, če velja:

- da ima edinstveno ime, ki ga poznajo vsi procesi, ki se sinhronizirajo med seboj,
- katerikoli od teh procesov ima dostop do sinhronizacijskega osebka v vsakem trenutku /LeLa81/.

Nekateri sistemi so zgrajeni tako, da preživijo napake, ki se pojavijo pri centralnem sinhronizacijskem osebku. Predvidene so tehnike za reševanje ob napakah, ki izberejo nov sinhronizacijski osebek, ko pride do napake.

Vsak sinhronizacijski mehanizem, ki temelji na centralnem sinhronizacijskem osebku imenujemo centraliziran.

Ostale mehanizme imenujemo porazdeljene.

Sinhronizacijski mehanizmi so lahko realizirani v elementih strojne opreme računalnika, kot primitivi v programskih jezikih ali v operacijskih sistemih.

5. ELEMENTI STROJNE OPREME ZA PODPORO SINHRONIZACIJE

Eden od najbolj enostavnih mehanizmov, ki omogočajo nedeljivost in medsebojno izključevanje procesov je onemogočanje prekinitev procesorja. Ta rešitev je bila uporabljena že na enoprocesorskih sistemih, ki so delovali na principu kvazi-paralelnosti. Na porazdeljenih sistemih nima posebnega pomena, saj procesorji med seboj ne morejo izvajati takih ukazov /Fink86/.

V tesno sklopljenih sistemih, kjer procesorji opazujejo in popravljajo vrednosti spremenljivk, je za ohranjanje konsistentnosti nujno zagotavljanje nedeljivosti branja in pisanja vrednosti spremenljivk glede na prebrano vrednost. Primera takih ukazov sta test-and-set in compare-and-swap /Hwan85/. Taki ukazi omogočajo realizacijo kompleksnejših sinhronizacijskih mehanizmov na višjem nivoju.

Obstaja še drug sinhronizacijski primitiv, ki je običajno realiziran v strojni opremi in dopušča določeno stopnjo sočasnosti pri uveljavljanju zaporednosti dostopa do skupnega pomnilnika. Imenuje se fetch-and-add. Primitiv ima obliko F&A(X,e) in vrne vrednost spremenljivke X ter poveča njeno vrednost za e. Če se izvede več takih ukazov hkrati, se vrednost spremenljivke poveča naenkrat za vsoto vseh e, vsak proces pa dobi vrnjeno vrednost, kot da bi se ukazi izvajali v naključnem vrstnem redu zaporedno.

V šibko sklopljenih sistemih uporabljajo kot osnovo za realizacijo nekaterih višjenivojskih mehanizmov posebne števnike, imenovane fizične ure. /Kope87/ predstavlja posebno časovno sinhronizacijsko enoto, ki poleg podatkov o realnem času vsebuje tudi mehanizme za usklajevanje in popravljane časovne baze, zasnovane na /Lamp78/. S tako enoto razbremenimo procesor, ki ga je izvajanje sinhronizacije preveč obremenilo.

6. SINHRONIZACIJSKI PRIMITIVI V PROGRAMSKIH JEZIKIH

Pomembna lastnost aplikacijske in sistemske programske opreme, ki povečuje preglednost in zmanjšuje kompleksnost, je njena modularnost. Navzven porazdeljen sistem tako izgleda kot množica programskih modulov , kjer je vsak modul zase enostaven zaporeden proces.
Procesi v tesno sklopljenih sistemih sodelujejo med seboj tako, da komunicirajo preko skupnih spremenljivk.

Osnovna rešitev problema medsebojnega izključevanje procesov, ki komunicirajo preko skupnih spremenljivk, je Dekkerjev algoritem /BenA83/. V njem so združene vse dobre lastnosti enostavnejših rešitev, kot so aktivno čakanje z opazovanjem skupne spremenljivke (busy-waiting) in uporabe spremenljivke za izmeničen dostop (switch-variable). Hkrati pa odpravlja pomanjkljivosti, zaradi katerih so te rešitve v praksi neuporabne. Mislimo predvsem na neodslednost pri medsebojnem izključevanju in na veliko odvisnost med procesi pri uporabi takega mehanizma.
Dekkerjev algoritem je precej kompleksen in ni primeren za implementacijo.

Veliko bolj enostavna rešitev je semafor /Dijk68/, ki ga je lahko implementirati in je dovolj močan, da lahko z njim elegantno rešimo probleme medsebojnega izključevanja in razvrščanja procesov. Dobra lastnost semaforjev, ki jo prej omnejeni mehanizmi nimajo, je tudi to, da so procesi, ki čakajo na doseg do določenega vira, blokirani. Zato medtem, ko čakajo, sprostijo procesne zmogljivosti za druge procese. Semaforje lahko uporabimo tudi pri implementaciji močnejših primitivov.
Semafor s je strogo pozitivna celoštevilska spremenljivka, ki ji je pridružena še vrsta

procesov.
Definirani sta dve operaciji nad semaforjem, P(s) in V(s), ki sta nedeljivi. Operacija P(s) zmanjša vrednost s za ena, če je s > 0, sicer pa odloži proces, ki je izvedel to operacijo v vrsto. Operacija V(s) pa zbudi prvi proces v vrsti, če pa je vrsta prazna, poveča vrednost s za ena.

Kritična področja odpravljajo edino slabo lastnost semaforjev, to je velika možnost za napake pri njihovi uporabi. Nepravilna razvrstitev operacij P in V je lahko usodna. Kritična področja so deli programa, kjer proces dosega skupne spremenljivke, ki so v tistem času nedostopne drugim procesom. Začetek in konec kritičnega področja označujejo posebni jezikovni konstrukti, ki so zelo enostavni za uporabo. Operacijski sistem potem sam poskrbi za dejansko realizacijo medsebojnega izključevanja.

Pogojna kritična področja so razširitev prejšnjega konstrukta. Vstop v krtično področje je dovoljen šele, ko je izpolnjen nek pogoj. Ovrednotenje pogoja in izvedba kritičnega področja sta nedeljiva.

Hoare in Brinch Hansen sta definirala monitor /Hoar74/, /BrHa73/. Podobno kot proces predstavlja koristno abstrakcijo pri mutiprogramiranju je monitor abstrakcija za medprocesno komunikacijo.
Monitor je razširitev pogojnih kritičnih sekcij. Monitor predstavlja telo, ki vsebuje skupne spremenljivke in procedure s *kritičnimi sekcijami za delo z njimi. S tem* postanejo te spremenljivke lokalne, skrite znotraj monitorja. Procesi, ki želijo dostop do takih spremenljivk, ga lahko dobijo samo preko monitorskih procedur. Monitor je pasiven v sistemu in se aktivira samo takrat, ko procesi želijo dostop do njegovih spremenljivk.

Pri šibko sklopljenih sistemih pa je sodelovanje med procesi v različnih programskih modulih povezano s pošiljanjem in sprejemanjem sporočil, ki služijo za sinhronizacijo in prenašanje podatkov /Slom87/. Komunikacija med procesi je lahko enosmerna ali dvosmerna.

Osnovna primitiva pri enosmerni komunikaciji sta 'pošlji' in 'sprejmi', pri dvosmerni pa zahtevek z odgovorom (request-reply), oddaljeni klic procedure (remote procedure call) in rendezvous.

Primitive za sinhrono sprejemanje podatkov najdemo v večini jezikov, ki so primerni za porazdeljene sisteme (ADA /USAD80/, CONIC /Slom85/, CSP /Hoar78/, SR /Andr81/, Pascal-m /Abra83/).
Modul, ki čaka na sprejem sporočila od drugega modula se blokira ter postavi v vrsto čakajočih in se aktivira šele po prejemu pričakovanega sporočila.
CSP ima podoben konstrukt tudi za pošiljanje sporočil.

Vsi dvosmerni primitivi omogočaju sinhronizacijo med procesi, ki jih izvajajo, čeprav se v podrobnostih in načinih izvedbe nekoliko razlikujejo.
Zahtevek z odgovorom je dvosmerni primitiv, ki je v bistvu kombinacija sinhronega pošiljanja in sprejema sporočila.

Oddaljeni klic procedure ima določene prednosti, saj omogoča transparentnost lokacije virov in procesov /Stan82/. To pomeni, da uporabniku ni treba vedeti, kje se nahajajo iskani viri niti mu ni treba sestavljati sporočil. Vse to zna narediti operacijski sistem.
Konstrukt v jeziku Ada /USAD80/, imenovan rendezvous, je kombinacija RPC in izmenjavanja sporočil. Omogoča sinhronizacijo procesov v času izvajanja konstrukta /BenA82/.

7. SINHRONIZACIJSKI MEHANIZMI V OPERACIJSKIH SISTEMIH

7.1. Centralizirani sinhronizacijski mehanizmi

Skupna lastnost centraliziranih sinhronizacijskih mehanizmov je v tem, da temeljijo na enem sinhronizacijskem osebku.

Fizična ura predstavlja osnovo za razvrščanje dogodkov podobno kot v centraliziranih sistemih. Operacijski sistem za realizacijo tega mehanizma uporablja posebne elemente strojne opreme. Procesi so razvrščeni na osnovi časovnih značk, ki jih dobijo, kadar je potrebna sinhronizacija.
Čeprav je ta metoda enostavna, ima mnogo pomanjkljivosti. Pravilno zaznamovanje dogodkov s časovnimi značkami je popolnoma odvisno od sprejema stanja ure ob vsakem dogodku. Napaka pri prenosu sporočila s tem podatkom je lahko usodna za pravilno razvrstitev. Potrebujemo tudi vnaprejšnje točno poznavanje zakasnitev v prenosnih kanalih. Natančnost je odvisna od zahtev sistema oziroma aplikacije.

Števec dogodkov je objekt, ki šteje dogodke, ki so se zgodili v določenem razredu (npr. aktivnosti). Definirani so trije primitivi: povečaj vrednost števca, preberi vrednosti števca in odloži klicoči proces, dokler ni vrednost števca vsaj enaka podani konstanti.
Pomembna prednost tega mehanizma je v tem, da dovoljuje sočasno izvajanje teh primitivov na istem števcu brez medsebojnega izključevanja. Pojem "števec z enim upravljalcem" je definiran kot števec dogodkov, ki deluje pravilno, dokler je prepovedano sočasno povečanje vrednosti števca. Vendar pa je sistem zelo občutljiv na izpad posameznih elementov, saj je vsaka napaka števca usodna za pravilno razvrščanje.

Statični razvrščevalnik uporabljamo za popolno razvrstitev dogodkov v danem razredu (števec dogodkov omogoča samo delno urejenost). Razvrščevalnik je celoštevilčna spremenljivka. Za delo z razvrščevalnikom je definiran samo en primitiv, imenovan vstopnica (ticket), ki vrne tekočo vrednost razvrščevalnika in poveča njegovo vrednost za 1. Razvrščevalnik zahteva ločen mehanizem za medsebojno izključevanje zato, da je primitiv vstopnica nedeljiv. Dva proizvajalca ne moreta dobiti vstopnice hkrati. Glavni problemi pri uporabi razvrščevalnika so pri izbiri mehanizma za medsebojno izključevanje in vpliv napak ali izpada razvrščevalnika na preživetje sinhronizacijskega mehanizma.

Jasno je, da centralizirani pristopi v porazdeljenih sistemih ne izpolnjujejo zahtev, ki so bistvene za take sisteme. Sistemi, zgrajeni na takih osnovah, ne morejo imeti visoke stopnje razpoložljivosti in imajo v splošnem slabše zmogljivosti zaradi

ozkega grla, ki ga predstavlja centralna enota.

7.2. Porazdeljeni sinhronizacijski mehanizmi

S porazdeljenimi mehanizmi dosegamo večjo stopnjo paralelnosti in s tem hitrejše delovanje, boljšo izkoriščenost opreme in večjo zanesljivost. Porazdeljeni sinhronizacijski mehanizmi so večkratne fizične ure, večkratne logične ure, abstraktni izrazi, skupne spremenljivke, krožeči žeton in krožeči razvrščevalnik.

Cilj uporabe fizičnih ur je v določitvi enotnega fizičnega časa v sistemu. Konsistentno dodeljevanje lahko dobimo iz popolne kronološke razvrstitve aktivnosti, ki se pojavljajo v sistemu. Pri uporabi več ur ni pomembna samo točnost vsake posamezne ure ampak tudi medsebojna usklajenost, tako da je razlika med poljubnima dvema urama manjša od vnaprej določene konstante. Rešitev tega problema je podal Lamport /Lamp 78/.
Sistem modeliramo kot povezan graf procesov s premerom d. Premer grafa predstavlja minimalno število procesnih mest, preko katerih mora iti sporočilo iz poljubnega procesnega mesta, da doseže poljubno drugo procesno mesto. Vsak proces ima uro in periodično (perioda t) pošilja sinhronizacijska sporočila vsekemu drugemu procesu. Vsako sinhronizacijsko sporočilo vsebuje fizično časovno značko. Po prejemu sinhronizacijskega sporočila proces pomakne svojo uro naprej, če je časovna značka večja od stanja ure. Predpostavljamo, da poznamo spodnjo mejo (u) in zgornjo mejo (u + z) zakasnitev na komunikacijski mreži. Naj bo k natančnost vsake ure ($k<10^{-6}$) in e dovoljen zamik med poljubnima dvema urama. Če je $e/(1-k) < u$ in $e << t$, potem je možno izračunati približno vrednost e, ki je približno d(2kt + z).
V odvisnosti od zahtev glede relativnih zamikov in veljavnosti predpostavk glede zakasnitev na komunikacijski mreži se lahko odločimo za tveganje, da bomo občasno izgubili sporočila zaradi prevelikih zakasnitev in tako dosegli verjetnostno sinhronizacijo ali pa ne bomo tvegali. Tak pristop bistveno zmanjša zmogljivosti sistema. Ključni parameter pri tem je razmerje z/u.

Večkratne logične ure so prvič opisane v /Lamp 78/. Implementirane so kot funkcije C, ki dodelijo število vsaki začeti lokalni aktivnosti. To so torej navadni števci. V sistemu, kjer ima vsak proizvajalec svojo logično uro je problem, kako zagotoviti globalno razvrstitev. Funkcija C ima lastnost, da velja $C(i,a) < C(j,b)$, če sta a in b aktivnosti v procesih i in j in velja a > b. Pri realizaciji logičnih ur moramo upoštevati dve pravili:

Pravilo 1: Vsak proces i poveča vrednost števca C(i) med dvema zaporednima aktivnostima.

Pravilo 2: Če aktivnost a v procesu i pošilja sporočilo in potem sporočilo m vsebuje časovno značko T(m) = C(i,a). Po prejemu sporočila m proces j postavi svoj števec C(j) na vrednost, ki je večja ali kvečjemu enaka trenutni vrednosti in je večja od T(m).

Vsako funkcijo C, ki ima zgoraj navedeno lastnost, lahko uporabimo za popolno razvrstitev poljubne množice aktivnosti. Za to potrebujemo še popolno ureditev procesov (npr. glede na njihova imena). Aktivnost a se je zgodila pred b, če je $C(i,a) < C(j,b)$ oziroma $C(i,a) = C(j,b)$ in je $i < j$. Sihronizacijski mehanizem definiran s pravili 1 in 2 in popolna razvrstitev dovoljujeta konsistentno dodeljevanje aktivnosti. Definirana popolna razvrstitev ni edinstvena in ni ekvivalentna kronološki razvrstitvi. To je razlog, da je včasih treba implementirati sistem logičnih ur na sistemu fizičnih ur.

Mehanizem z uporabo abstraktnih izrazov /Herm83/ temelji na uporabi logičnih ur. Vsaka ura ima poleg osnovne verzije, ki je pri procesu, ki povečuje njeno vrednost še dodatne verzije pri drugih procesih, ki samo spremljajo njeno vrednost. Predpostavimo, da mora vedno veljati abstrakten izraz

$$\sum c_i x_i < k$$

kjer sta c_i in k konstanti, x_i vrednost logične ure in i število logičnih ur.
Dodatne verzije logične ure imenujemo njene slike. Označene so z $m(x_i)$ in se lahko razlikujejo od osnovne verzije. Z uporabo dodatnih verzij pri delu s sinhronizacijskimi spremenljivkami zelo zmanjšamo komunikacijske potrebe in povečamo učinkovitost.
Zamenjava osnovne verzije logične ure z njeno sliko je dopustna, če pri zamenjavi upoštevamo tako stroge omejitve, da je kljub dopustni napaki slike izpolnjena zahteva osnovnega abstraktnega izraza.
To pomeni, da lahko pri slikah logičnih ur, kjer je konstanta c_i negativna popravljamo njihove vrednosti z zakasnitvijo, ne da bi s tem ogrozili pravilnost abstraktnega izraza. Podobno lahko vrednosti slik logičnih ur, ki imajo konstanto c_i pozitivno, popravljamo vnaprej.

Sinhronizacijski mehanizmi, ki temeljijo na fizičnih ali logičnih urah imajo skupno lastnost, da ne temeljijo na medsebojnem izključevanju. To je velika prednost pri uporabi v porazdeljenih sistemih, saj omogoča paralelnost izvajanja..

Sinhronizacijski mehanizmi lahko pri svojem delu uporabljajo dejstvo, da imajo procesi edinstvena in stalna imena. To določa popolno razvrstitev procesov, ki daje opazovalcu občutek, da so procesi povezani v neko verigo ali obroč. Vsak proces ima natančno določenega prednika in naslednika. Taka logična razvrstitev ni nujno povezana s fizično topologijo sistema.
Obroč služi kot osnova za prenašanje posebnih pravic med procesi v sistemu. Pri sinhronizaciji taka posebna pravica pomeni dostop do sinhronizacijskega osebka.

Skupne spremenljivke: Sinhronizacijski mehanizem, ki temelji na konceptu logičnega obroča je bil predstavljen v /Dijk74/. Ker predvideva uporabo skupnih spremenljivk je uporaben v tesno sklopljnih sistemih.
Lastništvo nad posebno pravico lahko zaznamo z opazovanjem spremenljivke, ki jo proces deli z enim od obeh sosedov v obroču. Z znanimi algoritmi lahko dosežemo stabilno stanje, kjer od večih posebnih pravic ostane samo še ena, ki potem kroži po obroču in zagotavlja medsebojno izključevanje.

Odkrivanje napak, reševanje iz napak in dinamično širjenje sistema so možni z metodami, opisanimi pri podajanju žetona.

Krožeči žeton je sinhronizacijski mehanizem zasnovan na enakih osnovah kot deljene spremenljivke, le da se posebna pravica prenaša med procesi s sporočilom, ki ima edinstveno obliko in ga imenujemo žeton. S tem principom dosežemo medsebojno izključevanje procesov v šibko sklopljenih sistemih. Po prejemu žetona lahko proces opravi želeno aktivnost. Žeton lahko obdrži največ nek naprej določen čas in potem ga mora predati naslednjiku v obroču. Mehanizem mora zagotoviti obnovitev logičnega obroča in podajanja žetona, če izpade trenutni lastnik žetona ali katerikoli element logičnega obroča, hkrati pa lahko dopušča v sistemu samo en žeton.

Protokoli za odkrivanje napak in njihovo reševanje so podani v /LeLa77/ in v /LeLa78/. Tak sinhronizacijski mehanizem je uporabljen tudi na logičnem nivoju lokalnih mrež in definiran v /ISO802/.

Učinkovitost sinhronizacijskih mehanizmov za medsebojno izključevanje med procesi je v veliki meri odvisna od časovne obsežnosti kritičnih sekcij. Če kritične sekcije vključujejo izmenjavo sporočil med proizvajalci in potrošniki, potem to daje nizke zmogljivosti.

Krožeči razvrščevalnik je razvrščevalnik, ki stalno kroži po logičnem obroču in uporablja mehanizem podajanja žetona.

Po prejemu žetona lahko proces aktivira več primitivov tipa vstopnica in potem pošlje žeton naslednjiku. Na ta način dosežemo medsebojno izključevanje uporabnikov razvrščevalnika. V članku /LeLa78/ je opisani protokol, ki uporablja opisani koncept.

Protokol uporablja številko obhodov, ki jo nosi žeton in se poveča vsakič, ko doseže proces, katerega naslednjik ima nižjo številko, kot je njegova.

8. KRITERIJI ZA VREDNOTENJE SINHRONIZACIJSKIH MEHANIZMOV

Sinhronizacijski mehanizmi nimajo identičnih lastnosti. V nadaljevanju podajamo važnejše kriterije za vrednotenje sinhronizacijskih mehanizmov. Pomembnost vsakega od njih je odvisna od omejitev, ki jih postavimo sistemu.

Odzivni čas in propustnost: Vsak mehanizem mora v največji možni meri upoštevati in izkoriščati vse prednosti paralelnosti delovanja porazdeljenega sistema. Paralelnost v sinhronizaciji in pri komuniciranju omogoča veliko propustnost in kratke odzivne čase.

Prožnost pomeni, da mora sinhronizacijski mehanizem biti odporen na pojav napak in zmanjšanja števila procesov v sistemu. Potrebujemo pravzaprav bolj natančno merjenje te lastnosti, ki bi izrazila število hkratnih napak oziroma izpadov procesov, ki jih sistem še preživi.

Dodatna poraba virov (overhead) je pravzaprav cena, ki jo moramo plačati, da lahko izvedemo sinhronizacijo. Kaže se v povečanju prometa med procesi in je odvisna od števila in velikosti paketov, v večji potrebi za procesiranje (dodatnih sporočil za sinhronizacijo) in v uporabi pomnilnika za shranjevanje potrebnih informacij.

Konvergentnost in poštenost sta lastnosti sistema, ki zagotavljata, da se v konfliktnih situacijah viri in zmožnosti enakomerno porazdelijo med tekmujoče procese.

Razširljivost je zahteva, da mora sinhronizacijski mehanizem dopuščati dodajanje ali ponovno vključevanje procesnih elementov, ne da bi s tem motil delovanje sistema.

Določenost. Sistem imenujemo deterministično, če je zasnovan tako, da v vsakem primeru doseže sinhroniziranost. Če pa doseže sinhronizacijo samo v večini primerov, se sistem imenuje verjetnosten.

Reševanje iz napak je zelo pomemben nabor funkcij. Posamezni mehanizmi se razlikujejo po količini pomoči, ki jo nudijo pri reševanju iz napak, po vplivu pokvarjenih elementov sistema na pravilno delujoče in po času, ki je potreben, da se popravljeni elementi spet vključijo v normalno delovanje sistema.

Povezanost med elementi sistema se zelo razlikuje glede na različne mehanizme. Slabi so taki mehanizmi, ki zahtevajo polno povezanost, saj je le-ta zelo draga v primerjavi s tipi povezanosti, kjer so elementi združeni v verigo ali obroč.

Vzpostavitve začetnega stanja se po kompleksnosti lahko zelo razlikujejo.

Razumljivost in enostavnost mehanizma nam olajšata delo pri snovanju, formalnem preverjanju, implementaciji, testiranju in vzdrževanju.

9. LITERATURA

/Andr81/ G.R.Andrews: The Distributed Programming Language, Software Practice and Experience vol.12 1982

/BenA83/ M. Ben-Ari: Principles of Concurrent Programming, Prentice Hall International 1983

/BrHa73/ P.Brinch Hansen: Operating Systems Principles, Prentice Hall 1973

/Dijk68/ E.W. Dijkstra: Cooperating Sequential Processes in Programming Languages, Academic Press 1968

/Dijk74/ E.W. Dijkstra: Self-stabilizing Systems in Spite of Distributed Control, Comm. ACM vol.17 no.11 1974

/Fink86/ R.A. Finkel: An Operating SystemsVade Mecum, Prentice-Hall 1986

/Herm83/ D. Herman: Towards a Systematic Approach to Implement Distributed Control of Synchronization, in Distributed Computing Systems, Academic Press 1983

/Hoar74/ C.A.R. Hoare: Monitors: An Operating Systems Structuring Concept, Comm. ACM vol.17 no.10 1974

/Hoar78/ C.A.R. Hoare: Communicating Sequential Processes, Comm. ACM vol.21 no.8 1978

/Hwan85/ K. Hwang, F.A. Briggs: Computer Architecture and Paralel Processing, McGraw-Hill 1985

/ISO802/ ISO IS 8802/4 Local Area Networks, Token Passing Bus Acces Protocol

/Kope87/ H. Kopetz, W. Ochsenreiter: Clock Synchronization in Distributed Real-Time Systems, IEEE Trans. on comp., vol.36 no.8 1987

/Lamp78/ L. Lamport: Time, Clocks, and the Ordering of Events, Comm. ACM, vol.21 no.7 1978

/Lmps81/ L. Lampson: Atomic Transactions, LNCS 105, Springer Verlag 1981

/LeLa77/ G. LeLann: Distributed Systems - Towards a Formal Approach, Proc.IFIP Congress Toronto, North-Holland 1977

/LeLa78/ G. LeLann: Algorithms for Distributed Data Sharing System which Use Tickets, Proc. 3.Berkley Workshop, August 1978

/LeLa81/ G. LeLann: Synchronization, LNCS 105, Springer verlag 1981

/Slom85/ M. Sloman et. al: The CONIC Toolkit for Building Distributed System Proc. 6.IFAC Workshop on Distr. Contr. Comp. Sys., Pergamon Press 1985

/Slom87/ M. Sloman, J. Kramer: Distributed Computer Systems and Computer Networks, Prentice Hall International 1987

/Stan82/ J.A. Stankovic: Software Communication Mechanisms: Procedure Calls vs. Messages, IEEE Computer, April 1982

/Trip87/ A.Tripathi: Distributed Operating Systems, Tutorial of 7th ICDCS Berlin, september 1987

/USAD80/ USA Departement of Defence: Reference Manual for the ADA Programming Language, Proposed Standard Document 1980

/Verj83/ J.P. Verjus: Synchronization in Distributed Systems, in Distributed Computing Systems, Academic Press 1983

MATEMATIČNO SODILO O MOŽNOSTI STROJNE INTELIGENCE (ali o Churchevem teoremu)

UDK 519.713.6

Lucijan Vuga
Iskra Delta, Nova Gorica

Church je 1936. leta izdelal teorem, ki ima globok vpliv na filozofske osnove kibernetike in na razrešitev dileme ali je mogoče izdelati inteligenten stroj.

Vprašanje: Ali lahko človek predvidi vselej in za vse primere, kakšen bo odgovor stroja, če vnesemo vanj ustrezen program in mu postavimo primerno vprašanje?

Odgovor, po Churchovem teoremu, ki gradi na diagonalnem postopku stroja, je lahko nepredvidljiv - vendar ne zaradi nakljućnih napak v HW ali SW, ampak zaradi matematično - logičnega mehanizma diagonalnega postopka, ki omogoča poleg znanih tudi nove kombinacije elementov, kar imenujemo ustvarjalnost.

Churc made in 1936 the theorem, that has deep influence on the philosophycal bases of cybernetics and the solution of dilemma whether is possible to make an intelligent machine.

The question: Could one foresee allways and for all instances, what will be the answer from machine, if inserts an adequate program and puts an appropriate question?

The answer, according to Church's theorem, which builds on the diagonal machine procedure, could be anpredictable - but not beacouse of accidental HW or SW faults, however owing to mathematical - logical mechanism of the diagonal procedure, that enables besides known also new combinations of elements, what we call creativity.

Človekov mladić si pridobi najosnovnejše pojme o geometriji s šestimi leti, z devetimi leti pa je sposoben, da manipulira z razdaljami v prostoru - in po trditvah nekaterih proučevalcev - odrasel šimpanz nić ne zaostaja za njim. Daleć od tega, da bi enačili človekove in opićje intelektualne zmogljivosti, toda že iz enega primera je videti, da so potrebna človeku dolga leta učenje, preden je sposoben kolikor toliko uporabljati najspecifićnejši organ - možgane. Torej, kot je znano, niso dovolj le možgani, temveć jih je treba napolniti z ustreznim znanjem, podatki itd. ter jih tudi ustrezno "splošno kondicionirati" - kot vsakdanje pravimo, ustvariti osebnost, ki je nujen pogoj za ustvarjalno delo. Pogoj pravimo zato, ker sicer brez tega manjka možganom odločitveni kriterij in dinamika vzgiba. Bogatenje duševnosti omogoća za ustvarjanje potrebno iniciacijo in za labilna, zamegljena stanja potrebno odločitveno nagnjenost.

Vendar se še vedno, tudi v načelu bije bitka, ali je sploh mogoče strojno oponašati delovanje človekovih možgan, ne oziraje se pri tem še na to, če smo sploh sposobni dokopati se do tehnologij, ki zagotavljajo uporabnost takega stroja. Tudi nekateri načelni privrženci strojne inteligence mislijo, da je do njene ostvaritve še zelo daleć.

Zanimivo bo pogledati neki matematični prijem, ki skuša po svoje razrešiti dilemo ali je stroj zmožen, za človeka nepredvidenih rešitev - torej, ali stroj lahko preseneti človeka? Pri čemer seveda ne mislimo na napake med delovanjem, slabo programiranje ipd.

Diagonalni postopek

Z diagonalnim postopkom je Cantor dokazal, da je množica realnih števil neštevna in da je zato moć take množice večja od moći množice racionalnih števil. Dve množici imata enako moć ali sta ekvivalentni, če je mogoće njune elemente drugega drugemu paroma prirediti, moć množice izrazimo s kardinalnim številom.

Vzemimo kot ilustracijo poseben primer: (1) realna števila naj bodo le tista med nić in ena (kar se lahko nato posplošii), (2) privzamemo, da vsako realno število v tem intervalu lahko zapišemo kot binarno število s karakteristiko nić in neskončno dolgo mantiso (če od določenega mesta naprej nastopajo v mantisi le ničle, potem imamo opraviti z racionalnim številom). Izhajamo iz možnosti, da je mogoće paroma prirediti elemente množice naravnih števil in naše množice realnih števil. S tem oštevilćimo **vsa** realna števila na naslednji način:

št. 1 ima $0,A_{11}\ A_{12}\ A_{13}\ldots$
št. 2 ima $0,A_{21}\ A_{22}\ A_{23}\ldots$
št. 3 ima $0,A_{31}\ A_{32}\ A_{33}\ldots$
itd.

Ker smo se tako dogovorili, ima vsako število A_{ij} vrednost 0 ali 1 in bi lahko imeli tudi neki tak primer:

1	0,0	0	1...
2	0,0	1	0...
3	0,1	0	0...

itd.

Sedaj pa se odločimo, da si skombiniramo novo število tako, da ga sestavimo iz diagonale zgornje tabele:

$$0,A_{11}\quad A_{22}\quad A_{33}\ldots$$

oziroma bi to bilo v našem izbranem primeru:

$$0,0\quad 1\quad 0\ldots$$

vendar pa bomo vsako cifro invertirali ter tako dobili iz ničle enko in iz enke ničlo:

$$0,B_1 \quad B_2 \quad B_3 \ldots$$

kjer je $B_i=0$, če je $A_{ij}=1$ in $B_i=1$, če je $A_{ij}=0$. V zapisanem primeru bo število dobilo tole vrednost:

$$0,1 \quad 0 \quad 1 \ldots$$

Očitno se to število razlikuje od kateregakoli števila v prvotni tabeli.

Ker smo se pa dogovorili, da bomo oštevilčili **vsa** realna števila, z diagonalnim postopkom pa je vselej mogoče izpisati še eno dodatno realno število, ki ga ni najti med prejšnjimi, je očitno to protislovno. (Z druge strani pa lahko preslikamo množico naravnih števil na delno množico realnih števil med nič in ena tako, da uporabimo inverzna realna števila: 1, 1/2, 1/3, 1/4,... ampak slednje nas tu sedaj ne zanima.) Church je 1936. izdelal teorem, ki se po njem tudi imenuje, ima pa globok vpliv tudi na filozofske osnove kibernetike in, videli bomo kasneje, tudi na razrešitev dileme ali je mogoče izdelati inteligenten stroj.

Churchev teorem

Preden se lotimo samega teorema je prav, da se dogovorimo o nekaterih definicijah.

Prvo: Kaj je algoritem? V veljavi je več ekvivalentnih in točnih definicij, kaj je algoritem - dovolj je dokazati nek teorem za eno od njih in avtomatično velja za vse ostale. **V intuitivnem smislu razumemo algoritem kot navodilo za postopno izvrševanje nekih operacij na nekih objektih (znakih, črkah), da bi dobili nek rezultat.**

Množico (naravnih števil) imenujemo **definitno**, če obstaja algoritem, ki lahko odgovori na vprašanje, ali neko naravno število pripada tej množici.

Zato je umestno vprašanje ali je vsaka števna množica tudi definitna?

Množica je števna, če obstaja algoritem, predpis, program, ki postopoma proizvede vse njene elemente. Če torej imamo tak program ali smo tudi gotovi, da bo izpisano tudi neko konkretno število? Program je dokončni tekst (je determiniran), ki ga lahko do popolnosti proučimo in iz njega izvlečemo vso informacijo, zato, na prvi pogled izgleda, ne bi smelo biti nikakršnih zadržkov za izpis katerega koli števila.

Če vzamemo poljuben programski jezik (algol, basic itd.) lahko prepoznamo vsako njegovo besedo kot del programa ali pa kot njegov nedel, ker je točno podana sintaksa programiranja v enem ali drugem jeziku. Od tod izhaja, da lahko programe številčimo z naravnimi števili. Tako lahko sestavimo univerzalni algoritem U, katerega izhodiščni podatki vsebujejo par cifer: številko algoritma N in število X nad katerim naj izvajamo N-ti algoritem. Tako bo U vseboval navodilo, kako naj dešifrira število N in ga preoblikuje v tekst ustrezajočega programa, nato pa še tak algoritem izvede na številu X. Če lahko N-ti algoritem označimo z A_N, lahko vse skupaj napišemo v obliki:

$$U(X,N) = A_N(X).$$

Church je raziskal "diagonalni algoritem" U (X,X), ki deluje tako, da oponaša uporabo algoritma 1 na številu 1, nato uporabo algoritma 2 na številu 2, itd. Tako dobljena množica bo področje, ki ga definira algoritem U (X,X)+1, ki ga lahko krajše napišemo kot algoritem za število X, torej D(X).

Predpostavimo, da je področje, ki ga definira ta algoritem definitno, tj. da za vsako naravno število zna odgovoriti ali pripada ali ne pripada temu področju. Zato lahko sestavimo algoritem B(X) podan z enačbama:

$$B(X) \quad \begin{cases} D(X), \text{ če je D uporaben na X} \\ 0, \text{ če D ni uporaben na X.} \end{cases}$$

Kot vsak drugi algoritem ima tudi B neko številko, ki naj bo recimo M torej B(M), tedaj lahko napišemo B(X) = U(X,M). Od prej vemo, da je B(X) povsod definiran - zato je definiran tudi B(M) in s tem še U(M,M). Če pa je tako, je definiran tudi U(M,M)+1, ekvivalentno zgoraj opisanemu D(M). Po konstrukciji B velja B(M)=D(M)=U(M,M)+1. Z druge strani uvrstimo X=M z enačbo B(X)=U(X,M), dobimo B(M)=U(M,M). To pa je protislovno, kar pomeni da obstaja števna množica, ki pa ni definitna. To je Churchev teorem.

Uporaba Churchevega teorema na strojih

V.N. Trostnikov opozarja, da je mogoče Churchev teorem uporabiti za preiskavo možnosti izdelave inteligentnega stroja in navaja, da se običajno definira razliko med človekom in strojem na naslednji način:

"Obnašanje stroja je strogo determinirano, medtem, ko ima človek lastnost, ki ji pravimo svobodna volja."

Vendar kje naj bi se kazala ta svobodna volja, v kateri razvojni stopnji ustvarjalnega razmišljanja? Ni potrebno, da se strinjamo prav z vsem kar je za naslednjo shemo, ki jo uporabljajo nekateri psihologi, je pa primerna pri obravnavi te snovi, da je ustvarjalnost postopnost naslednjih dejavnosti: priprave, inkubacije, inspiracije in preverjanja.

Priprava naj bi zajemala kopičenje potrebnega znanja, podatkov ter ustrezno urejanje. **Inkubacija** za katero mnogi trde, da se odvija med določenimi podzavestnimi procesi: asociranjem, odpadanjem nekaterih idej po zakonu nedavnosti i.p. **Navdihnjenje - inspiracija**, ko se povežejo različni podatki, dejstva, elementi v novo kombinacijo. **Preverjanje** iz inspiracije nastale zamisli, ki terja zavestno, organizirano in sistematično delo z že določenim ciljem.

Izhajanje iz te sheme, bi lahko rekli, da je le v inspirativni fazi potrebna neka "svobodna volja", ki naj zagotovi novo, doslej neznano kombinacijo znanih elementov. Tako pripravljalna faza, kakor tudi inkubacija nimata sestavin ustvarjalnosti, kljub nekaterim neznanim dejstvom o podzavestnih procesih, saj gre za pripravo gradiva. Res je tudi, da pri tem zelo verjetno nastopajo medsebojna prekrivanja faz ter različne interakcije, povratne zveze i.p. Toda pravi trenutek stvaritve je zavedanje, da je nastala nova kombinacija, čemur pravimo inspiracija. Poslednja aktivnost, preverjanje, pa je sicer izredno pomembna, včasih dolgotrajna in naporna, toda ta deluje po ustaljenih principih, v kolikor seveda ni potrebno tudi za preverjanje ustvariti kaj novega, kar terja postopek podoben prejšnjemu - sicer seveda ostane nova zamisel brez potrditve zgolj hipoteza.

Oglejmo si zato nekoliko podrobneje inspirativno fazo. Trostnikov trdi,da bi morali definirati vprašanje o svobodni volji, ki naj bi bila osnova ustvarjalnemu mišljenju, nekoliko drugače, pri tem pokliče na pomoč tudi Turinga, ki je rekel, da bi morali "postati stroj, če bi hoteli vedeti kaj stroj čuti". Zato si je treba pomagati drugače z vprašanjem:"Ali se človek in stroj načelno razlikujeta v tem, da je obnašanje stroja, potem ko poznamo njegov program, v vsakem primeru (načelno) predvidljivo, medtem ko je pri človeku nepredvidljivo?" Ker smo že spoznali diagonalni postopek in Churchev teorem, ki sta nam pokazala, da kljub jasnemu algoritmu nismo nikoli prepričani - in to načelno, ne zaradi tehničnih napak v delovanju stroja - ali je ali ni proizvedeno število del množice, ki jo generira algoritem.
Zaključek tega razmišljanja je torej tak, da tako človek kakor stroj nista determinirana in imata ustvarjalne sposobnosti.

"Svobodna volja"

Seveda pri tem ostaja še vedno odprto vprašanje svobodne volje, ki naj bi bila posebna človekova sposobnost, da lahko pri razsojanju, hotenju in delovanju odloča in izbira med dvema ali več odločitvami. Filozofi-idealisti imajo človekovo svobodo za absolutno, kar pomeni, da ni odvisna od objektivne nujnosti naravnih zakonov, in je glede na to znak človekove pripadnosti nekemu nadnaravnemu svetu. Za materialiste je svoboda "nujnost, ki je prešla v zavest"; ne obstoji v namišljeni neodvisnosti od naravnih zakonov, ampak v spoznanju teh zakonov in na tem znanju temelječi možnosti izkoriščanja naravnih zakonov in sil v prid določenemu cilju; "svoboda volje pomeni sposobnost odločevanja na temelju poznavanja stvari" (Engels); "svoboda je spoznana nujnost" (Hegel). Izhajajoč iz slednjih opredelitev tudi človek pri ustvarjanju ni svoboden, čeprav izgleda, da je prav prebijanje omejitev, ki so bile "spoznane kot nujnost", ustvarjalni napor, je tudi res, da tudi nekonvencionalnost, iskanje novega, potreba po ukinjanju aksiomov itd., je za človeka "spoznana nujnost". Nujno je spoznavanje narave, nujno je reševanje neznank, nujno je iskanje novih kombinacij, v tem je osnovni vzgib ustvarjalnosti, torej tudi "nepredvidljivosti" - čeprav lahko vsaj načelno pričakujemo nova spoznanja, odkritja, izume.

Nedvomno je mogoče tudi stroju priskrbeti toliko izkustev, znanja, podatkov, da bo "spoznal nujnost" posameznih odločitev, dasi seveda pri tem ostaja vprašanje tehničnih možnosti kot rešljiv a zelo zahteven problem. Program, ki je potreben za delovanje stroja izdela človek, ni pa rečeno, da tega ne bi znal tudi stroj, če bi mu dali program za programe morda vsaj za posamezne dele prvotnega programa. In prav to je pomembno pri ustvarjanju "svobodne volje" samega stroja. Program vsebuje ukaze za računanje, naslove, odločitve i.p. To že pri programiranju razumemo dobesedno in pričakujemo,da bo stroj stoodstotno izvršil naš ukaz; če se tako ne izvrši smatramo, da je stroj pokvarjen ali da obstaja neka druga napaka, ki preprečuje dobesedno natančnost. Pri človekovem možganskem delovanju pa ni tako. Pojmi s katerimi operiramo so megleni, mehki, nenatančni. Za obravnavanje takih stanj, pojmov in postopkov je razvita teorija zamegljenih (mehkih) množic - fuzy sets theory. Tako n.pr. pri odločitvi: da - ne, nimamo ostrega in jasnega prehoda, temveč obstaja vrsta vmesnih stanj med 1 - 0 da-ne. Za oponašanje take situacije moramo dati stroju možnost da se s pomočjo podprograma, na osnovi vloženega znanja, izkušenj itd., opredeli za stopnjo določenosti med 0 in 1, med ne in da. Še več! Tak podprogram ne bo en sam, niti ne bodo taki podprogrami fiksni, temveč bodo odprti, dogradljivi, spremenljivi. S tem bomo dobili možnost za nedeterminiranost strojnega dela in za približevanje načinu možganskega funkcioniranja, pri čemer bo posamezna odločitev odvisna od kopičenja potrebnega znanja, podatkov, okoliščin itd.

Nedvomno lahko na tak način pridemo do adeterminističnega stroja, torej takega, katerega proizvodi ne bodo vselej pričakovani, saj bodo sledili funkcioniranju notranjega ustroja, katerega logika se bo lahko razlikovala od človekove, ki je zgrajena na zgodovinskem izkustvu konkretne realnosti v katerem poteka filogeneza človeštva. Tako kot nas sedanji stroji prekašajo po hitrosti, natančnosti in obsežnosti obdelav, lahko domnevamo, da nas stroji lahko prekašajo tudi pri drugih zmožnostih kombiniranja, analize i.p.

Tako pridemo do usodnega sklepa, če naj ima stroj vložene vse podatke, ki so potrebni za njegovo delovanje in odgovarjajoče funkcioniranje adeterminističnega dinamičnega programiranja, mora vedeti tudi, da sam obstaja: Cogito ergo sum. Poznati mora svoje sposobnosti, zmogljivosti, obseg obdelav, spomina, samoprogramiranja itd. ter tudi to, koliko je že obremenjen, na katerem delu so proste kapacitete, poznati mora stopnjo zasedenosti spomina i.p.- skratka zavedati se mora sam sebe. Tudi človek si z leti pridobiva samozavedanje z vzgojo in izobraževanjem, kar prinese na svet z rojstvom so zelo primitivne oblike zavesti. Človekova čustva in najrazličnejša duševna stanja so bodisi delna ali splošna podprogramska kondicija, ki ustvarja primerne pogoje za odločanje, zlasti, ko gre za zamegljena stanja okoli neodločne lege, za iniciacijo novih zamisli, ki pač ne nastajajo iz niča ali same od sebe, temveč zaradi vzdrževanja "duševnega tonusa", ki je individualno stanje možganskega sistema potrebno za zagotavljanje neapatičnosti, mrtvila. (V množičnih psihozah i.p. je mogoče močno poenotiti duševna stanja posameznih individuumov.) To bi nekako lahko primerjali z inkubacijsko stopnjo mišljenja. Podobno je mogoče ustvariti v stroju "programe za programe", katerih ena od nalog je prav ta, da ustvarja "duševno razpoloženje", ki je potrebno tudi stroju, ko črpa iz zalog znanja, podatkov, informacij, da črpa po nekem sistemu, tudi če je to naključni sistem, ter da kombinira znana dejstva v nova dejstva, čemur rečemo ustvarjanje t.j. inspirativna faza. Formalno je najmanj težav videti pri samem preverjanju nove zamisli - ampak preverjanju v "teoretičnem smislu", ko niso potrebne dodatne meritve ali celo ustvarjanje novih merilnih postopkov in naprav, da bi potrdili novo zamisel. Stroj bi lahko novote primerjal le s tistim, kar si je nabral v svojem pomnilniku. Vendar je prav enako tudi s človekom! Šele, ko si človek z meritvami pridobi dodatna dejstva, podatke, informacije in jih primerja z novimi zamislimi, ve, pri čem je. Prav verjetno je, da Churchev teorem omogoča ustvarjati, ko pojasnjuje načelno nedeterminiranost strojnega delovanja, take rešitve in računalniške zasnove, ki bodo približale strojno človekovi inteligenci, ter slednjo v specifičnem smislu tudi presegle.

Literatura:

1. Douglas R. HOFSTADTER: GOEDEL, ESCHER, BACH (ADELPHI Ed.), 1984, Milano

2. Gerald Jay. SUSSMAN: A COMPUTER MODEL of SKILL ACQUISITION, (AMERICAN ELSEVIER), 1975, N.York

3. Georg K. ZIPF: HUMAN BEHAVIOR and the PRINCIPLE OF LEAST EFFORT (ADDISON-WELSEY PRESS), 1949, Cambridge