YU ISSN 0350-5596

informatica

JOURNAL OF COMPUTING AND INFORMATICS

YU ISSN 0350–5596

VOLUME 9, 1985 – NO. 2

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

HEADQUARTERS: Informatica, Parmova 41, 61000 Ljubljana, Yugoslavia
Phone: 61-312-988; Telex: 31366 YU DELTA

ANNUAL SUBSCRIPTION RATE: US$ 22 for companies, and US$ 10 for individuals

PRINTED BY: Tiskarna Kresija, Ljubljana

DESIGN: Rasto Kirn

CONTENTS

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 9, 1985 – ŠT. 2

Časopis izdaja Slovensko društvo INFORMATIKA, 61000 Ljubljana, Parmova 41, Jugoslavija

UREDNIŠTVO IN UPRAVA: Informatica, Parmova 41, 61000 Ljubljana; telefon (061) 312-988; teleks 31366 YU Delta

LETNA NAROČNINA za delovne organizacije znaša 1900 din, za redne člane 490 din, za študente 190 din; posamezna številka 590 din.
ŽIRO RAČUN: 50101-678-51841

Pri financiranju časopisa sodeluje Raziskovalna skupnost Slovenije.

Na podlagi mnenja Republiškega sekretariata za prosveto in kulturo št. 4210-44/79, z dne 1.2.1979, je časopis oproščen temeljnega davka od prometa proizvodov

TISK: Tiskarna Kresija, Ljubljana

GRAFIČNA OPREMA: Rasto Kirn

V S E B I N A

INFORMATICA 2/85

MODELING AND ANALYSIS OF COMMUNICATION PROTOCOLS AND COMPUTER NETWORKS USING PETRI NETS

BOZIDAR BLATNIK

DO ISKRA DELTA, LJUBLJANA

UDK: 681.3:007

Abstract. Communication protocols intrinsically involve parallelism and therefore fall into the class of problems for which Petri nets have been defined. The methods developed for the specification, design and analysis of communication protocols using Petri nets are briefly discussed in this paper. A survey of work done in this area in the recent years is presented and referred to.

MODELIRANJE IN ANALIZA KOMUNIKACIJSKIH PROTOKOLOV IN RAČUNALNIŠKIH MREŽ S PETRIJEVIMI MREŽAMI. Petrijeve mreže so učinkovito orodje za modeliranje in analizo paralelnih sistemov. V članku so opisane nekatere metode iz teorije Petrijevih mrež, ki se uporabljajo za specifikacijo, načrtovanje in analizo komunikacijskih protokolov in računalniških mrež. Podan je pregled do sedaj opravljenega dela na tem področju in možne smernice za nadaljnje raziskave in razvoj.

1. Introduction

Petri nets are a class of mathematical models of the structure and dynamic behavior of nondeterministic systems. In a relatively short space of years Petri nets have become the dominant model of concurrency and parallelism in computer systems. They have been applied to many different areas of practical interest in computer science, including the design and analysis of pipelined hardware and the deadlock analysis of operating systems, inspired by the work of Hack et al on the Information System Theory Project during 1968 to 1971 and of Dennis et al on the Computation Structures Group of Project MAC during the same period. The original motivation for Petri nets was, however, to provide a rigorous basis for the study of communications.

Petri nets were first formulated and investigated by Carl Adam Petri in his 1962 doctoral dissertation ((1)). His opening paragraphs well-state his orientation in the project:

"This work is concerned with the conceptual foundations of a theory of communication. It shall be the task of this theory to describe in a consistent and exact manner as many as possible of the phenomena that occur in the transmission and transformation of information..."

Petri net theory has developed considerably since its beginnings in 1962. However, much of the work is hard to obtain, being available only as reports and dissertations scattered among many sources. The major parts of Petri net theory are brought together in a concise and consistent manner in ((2)). It is becoming expected that every computer scientist know some basics.

It has only been in the last ten years that Petri nets have actually been applied to the verification of communication and network protocols, corresponding no doubt to the greatly increased importance of these areas during this time, e.g. ((3)), ((4)), ((5)).

The distinction between Petri net models of communication protocols and network protocols and, thence, other applications is centered around the importance of time. In Petri's original critique of automata theory, the presupposition of the synchrony of deterministic automata was isolated as a fundamental stumbling block to their application to communications. Insisting that the automata must map to reality, he deduced that not only is communication a nondeterministic process but that the time delay in propagation of state transitions was considerable and that an automaton fundamentally dependent on synchrony could not model the process he was examining. Therefore, the basic principle on which he would reconstruct automata theory would be that synchronization decisions must be made purely locally at each state. Basic Petri nets are purely time-independent. Petri nets used in

network protocol applications are weakly-locally time-dependent, usually only presupposing the existence of a "time-out" timer which may be started when a state is entered and which will cause a transition to an "error" state if no other transition has occurred by the time the timer turns over. Communication protocol applications are locally time-dependent, usually presupposing a minimum-time-to-transition concept. It has been demonstrated that these concepts together with the intrinsic interpretation of Petri nets as logical state space diagrams can be profitably used to analyze protocols from the data link layer to the application layer in a network architecture.

Petri net models are used in two basic modes: simulation modeling and analytic modeling. These two modes will be discussed in the following two sections, focussing primarily on analytic modeling. The final section of this paper will briefly discuss possible directions for future study.

2. Simulation Modeling

Petri net simulation systems are suites of software tools which allow the input-output, manual manipulation, automatic execution, and partial automated analysis of Petri net models. The first known such system was implemented at the University of Washington by Noe et al. ((6)). This system consisted mainly of a graphics editor for the input-output of Petri nets and the firing of enabled transitions input by the user. More sophisticated systems are under development at various institutions in France and Germany. It is convenient to discuss the nature of Petri net modeling in this context.

A Petri net model consists of three components: a static substructure, a dynamic substructure, and an analytic substructure. The static substructure of a Petri net model is represented by a Petri net graph which represents the a priori relationships between the various states of the system being modeled. It is a bipartite directed graph. The two classes of nodes are called places and transitions, represented by circles and bars, respectively. The arcs of the digraph are restricted to join place-transition or transition-place pairs only. An example of such a construction representing a simple protocol for a transmitter (left side) and a receiver (right side) is given in Figure 1.

The most frequent interpretation of these digraphs in net theory is to consider the places as all pertinent predicates or states which can hold in the system and the transitions as all pertinent events which can occur in the system. The input places of a transition are the preconditions which must hold for the event to occur, and the output places of a transition are the postconditions which will hold after the event occurs.

Events are viewed as logical state transformers. When an event occurs, under the standard interpretation, the postconditions come-to-hold while the preconditions, simultaneously, cease-to-hold. This ebb and flow of the states of the system is regulated by the dynamic substructure. The dynamic substructure of a Petri net defines the idea of a marking of the places of the Petri net graph and defines a firing rule to govern the behavior of the transitions. A marking is an assignment of tokens, represented by dots, to places. The absence of a token in a place indicates that the corresponding predicate does not currently hold, while the presence of a token indicates that the predicate does currently hold. The canonical firing rule for Petri nets states that the firing of a transition subtracts one token from each of its input places and deposits one token into each of its output places. The only transitions which may be fired, however, are those which are enabled by having at least one token in each of its input places. If several

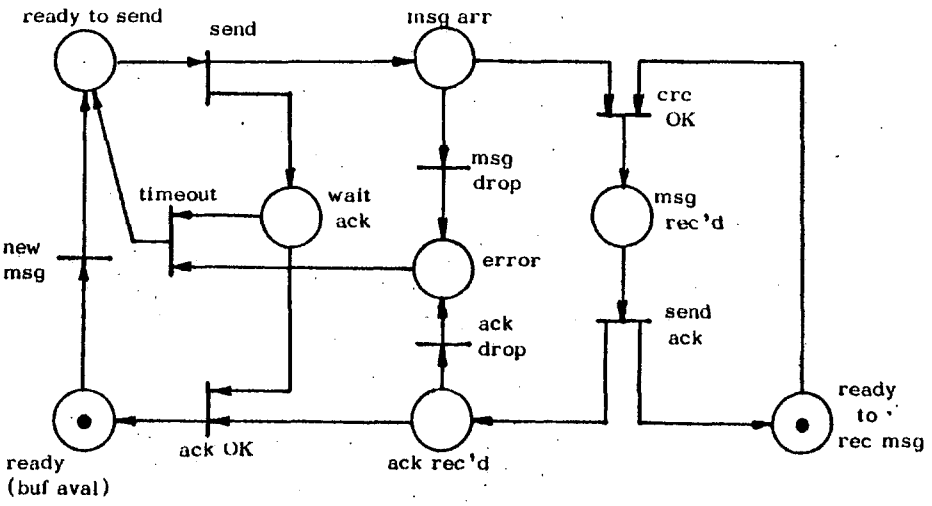


Figure 1. Petri net model of a simple transmitter-receiver protocol.

transitions are enabled, then one is chosen at random, for normal Petri nets.

The Petri nets used for protocol analysis, however, are not normal Petri nets. They are a special class of nets usually called time nets. Merlin was the first to do work on evaluating communication protocols using Petri nets ((7)). His work concentrated on physical telephone protocols and is an example of locally time-dependent nets. In this model, each transition has associated with it two times, t1≤t2. He postulates a local timer for each transition which starts when all the input places for the transition contain at least one token. The transition becomes enabled and fireable after time t1 has elapsed and must fire before time t2 has elapsed, all of which assuming that at least one token remains in all the input places throughout. If not, the timer is reset and stopped.

Other variations on time nets have been proposed. Coolahan and Roussopoulos ((8)) describe a variant not unlike Merlin's but only distinguish t1. This model is more analyzable than Merlin's, and Coolahan and Roussopoulos sketch their proof procedures, but still this technique remains basically a simulation model. In a different vein, Molloy suggests that allowing the transition delay time t1 for each transition to be exponentially distributed is of interest. For an interesting subclass of Petri net graphs, he shows that such exponential time nets are equivalent to ergodic Markov chains. Within this domain, the nets are analyzable by standard techniques and are easier to formulate in their Petri net form than in a Markov process form. Outside this form, however, recourse is made to simulation ((9)).

The last component of a Petri net model is the analytic substructure. This consists of the definitions of the properties of the model which are to be decided or quantified and the properties which are to be preserved under any net morphism which is applied to simplify the net. The three classic predicates of Petri net theory, namely liveness, safeness, and reachability, also have application to protocol models. A Petri net is live iff it cannot deadlock, i.e. iff for all reachable markings from some initial marking at least one transition is enabled. A net is safe at level k iff for all reachable markings from some initial marking the maximum number of tokens in any place is less than or equal to k. This property becomes important when it is realized that places may represent message buffers of finite capacity (if k is exceeded, then messages have been lost or overwritten) or when a place has a one-to-one correspondence to a message currently being processed (in which case, bookkeeping information about the message has been confused with the message following). A marking of a Petri net is reachable from a given initial marking iff there exists a sequence of enabled transitions whose firing would produce the marking. Reachability considerations become important when determining whether normal and exception states can ever be reached. For example, can the message transmitted ever be received using the protocol being studied?

The properties to be preserved under a net morphism ((10)) are varied and may be particular to an application. They include the properties above (liveness, safeness, reachability) as well as certain special ones defined by Merlin for protocol models. These latter properties are considered by Merlin to be necessary for all well-behaved protocol models. They are: the number of states is finite, from any marking reachable from the initial marking the initial marking is itself reachable (always cycle back to the initial marking), there is no direct loop from an exceptional state to an exceptional state (the protocol does not get lost on error conditions), and when one party in the communication link returns to an idle state then the other will return to its idle state within a finite amount of time (i.e. back to the initial state).

Certain other quantities to be computed have been defined by Molloy ((11)) and Ramamoorthy and Ho ((12)). These include: mean time in state, mean cycle time through the net, expected exception rates, and mean throughput. Although these authors demonstrate analytic techniques for deriving this information for special case nets, for the general cases simulation is still resorted to. Such simulation systems must include mechanisms to encompass the static and dynamic substructures of the Petri net model and should include automated tests, at minimum, for the failure of the properties just discussed. Ideally, though, tools which actually decide these properties should be included. These tools would be based on the techniques of the next section.

3. Analytic Modeling

The goal of analytic modeling is to develop finite deterministic decision procedures for the predicates of the analytic substructure which are effective. Petri net models for any practical-sized problem are of a complexity that is generally unreasonable to expect that the analysis will be carried through by human computation. Therefore, if the techniques are not automable, they are useless. This is so, though, with the understanding that several of the most interesting problems are provably NP-space-hard ((13)) or NP-complete ((12)).

The authors who have proposed time net models just discussed have included in their results analytic findings and in some cases have given proof methodologies to find the quantities or decide the predicates enumerated above. But, these techniques have been ad hoc, rather brute-force approaches automable only with the use of a general theorem prover. Recent advances in net theory have demonstrated the real utility of a linear algebraic approach to Petri net analysis, e.g. ((14)). It is this theory which will be briefly discussed here.

The basis of this technique is the incidence

matrix for the Petri net graph and linear invariants for this matrix. To begin, this theory applies to self-loop-free graphs, i.e. Petri net graphs where no place can be both an input place for a particular transition and an output place for that transition. For the purposes of modeling protocols, this is not a severe restriction. The incidence matrix N for a net is a p by t matrix where the i,j entry of N is 0 iff there is no arc from the i-th place to the j-th transition or from the j-th transition to the i-th place and is the difference between the number of i,j arcs and the number of j,i arcs otherwise, where p and t are the number of places and number of transitions, respectively. A marking m is a p by 1 vector representing the number of tokens in each place. A transition vector x is a t by 1 vector indicating which transitions are to be fired. The firing rule for Petri nets which produces a new marking m' is, then,

$$m' = m + Nx \ .$$

An invariant (place invariant) v is defined as the solution to the matrix equation

$$vN = 0 \ .$$

The central theorem of this technique states that

$$vm' = vm \ .$$

The proof is as follows:

$$\begin{aligned} vm' &= v(m + Nx) \\ &= vm + 0x \\ &= vm \ . \end{aligned}$$

The use of invariants is very lucidly presented by Jensen ((15)) using the classic readers-writers problem. In Figure 2, the states indicate local processing (LP), waiting to write (WW), writing (W), reading (R), and synchronizing (S).

The incidence matrix, initial marking and invariants are shown in Table 1.

	t1	t2	t3	t4	t5	t6	m0	i1	i2	i3
LP	-1	-1			1	1	n	1		-1
WR	1		-1					1		-1
WW		1		-1				1		-1
R			1		-1			1	1	
W				1		-1		1	n	(n-1)
S			-1	-n	1	n	n		1	1

Table 1. The incidence matrix, initial marking and invariants for the example in Figure 2. The set of places consists of LP, WR, WW, R, W, and S; the set of transitions consists of t1,...,t6; m0 is the initial marking; i1, i2, and i3 are invariants.

Applying the fundamental theorem for each invariant, we derive the equations:

$$\begin{aligned} &(i1) \quad m'(LP)+m'(WR)+m'(WW)+m'(R)+m'(W) = n \\ &(i2) \quad m'(R) + nm'(W) + m'(S) = n \\ &(i3) \quad m'(LP)+m'(WR)+m'(WW) = (n-1)m'(W)+m'(S), \end{aligned}$$

where $m'(X)$ is the number of tokens in the X-th place of marking m' and the right-hand sides are the results of vm0.

It follows from i1, that the number of processes is invariant. It can be deduced form i2, that if one process is writing, then no other process is writing or reading. From i3, however, little of immediate use is derivable since i3 is a linear combination of i1 and i2.

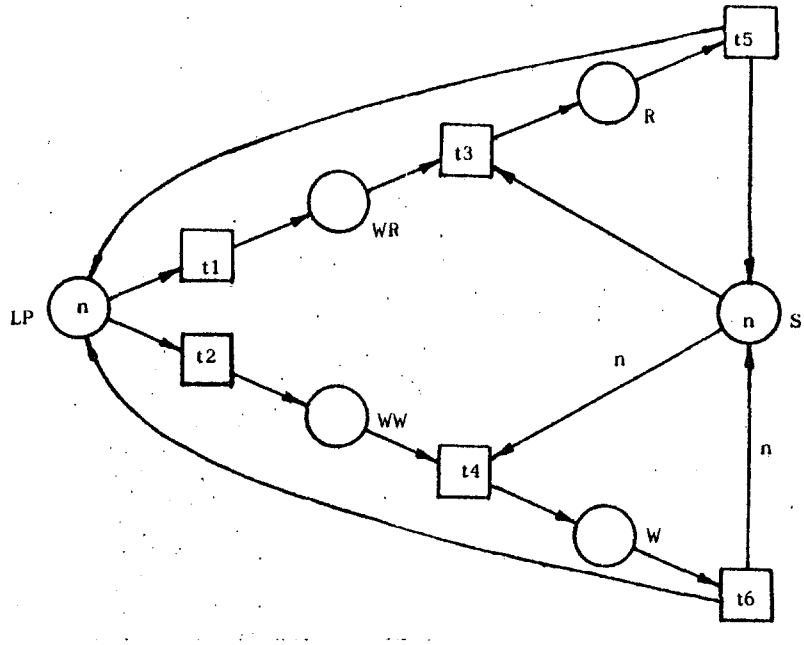


Figure 2. Petri net model for the readers-writers problem.

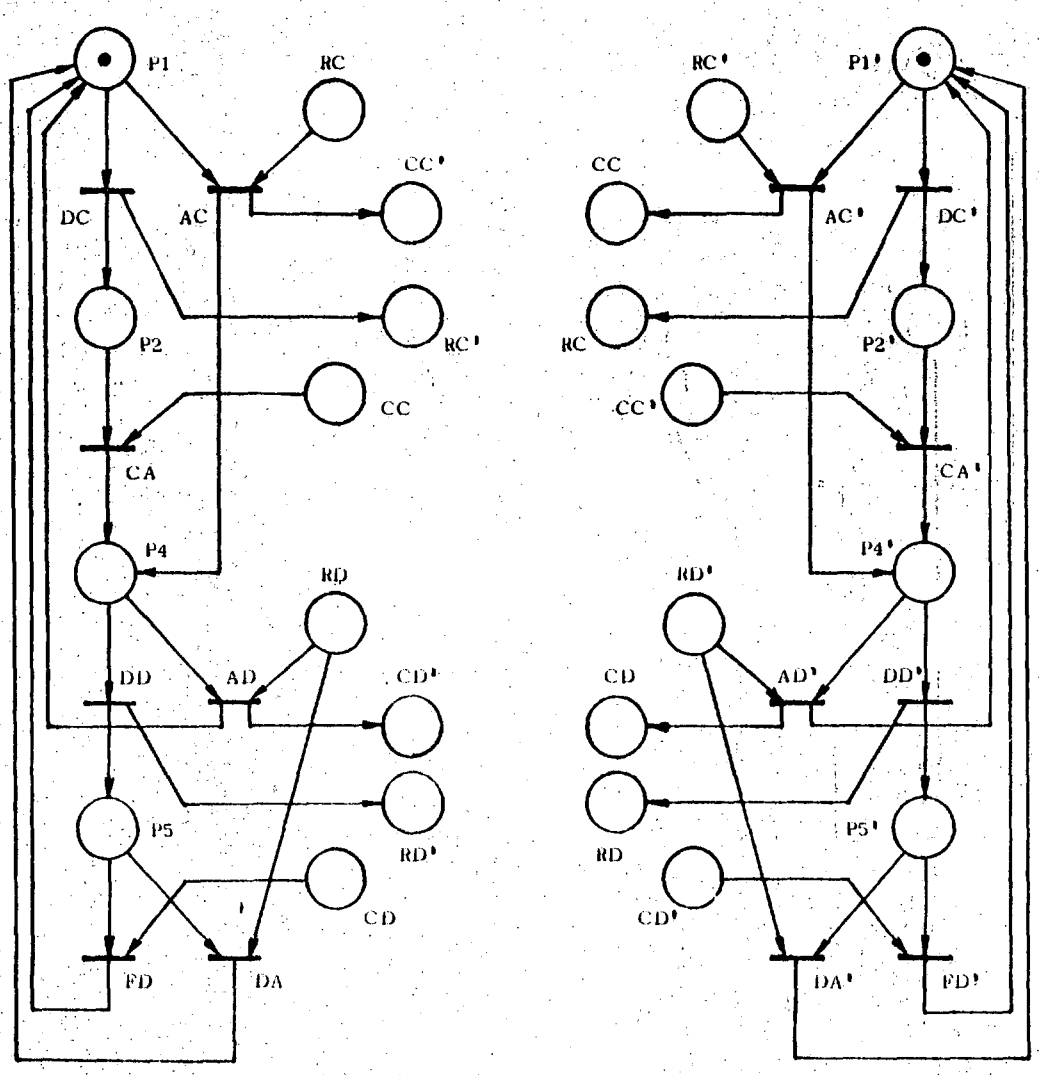


Figure 3. The connect and disconnect phases of the ECMA transport protocol standard (consonant with ISO standards).

The proof of the liveness of the net is as follows: i1 and i2 admit the possibility that $m'(LP)$, $m'(R)$, and $m'(W)$ are nonzero. If so, then they enable t1 and t2, t5, and t6, respectively. If not, then $m'(WR) + m'(WW) = n$ and $m'(S) = n$ which enables either t3 or t4 or both. At least one transition is always enabled, therefore the net is live. Proofs of safeness and other properties are substantially similar.

Berthelot and Terrat have applied these same techniques to validate the European Computer Manufacturers Association (ECMA) transport protocol standard which is consonant with ISO standards ((16)). The connect and disconnect phases of this protocol are represented in the net in Figure 3.

An additional technique they use is that of net reductions. First, trivially, like labeled places are coalesced into one place, thus bringing the two halves of the net together as one connected graph. Then, since there is one token in P2 iff there is one token in RC' and CC, P2 is redundant and may be removed. Since transitions DC, AC', and CA are serially fired, they can coalesced into AC', thus eliminating the need for RC' and CC. These net reductions were defined and studied by Berthelot, Roucairol, and Valk ((17)) and their application here provides an interesting demonstration of how Petri net analysis can improve a protocol specification. The reduced net, transition matrix, initial marking and invariants are given in Figure 4 and Table 2.

It is shown in their paper that this net is live, safe at level 1, and that the initial marking is a home-state, i.e. is reachable from every marking reachable from the initial marking. Thus, this protocol is well behaved in the sense which Merlin defined.

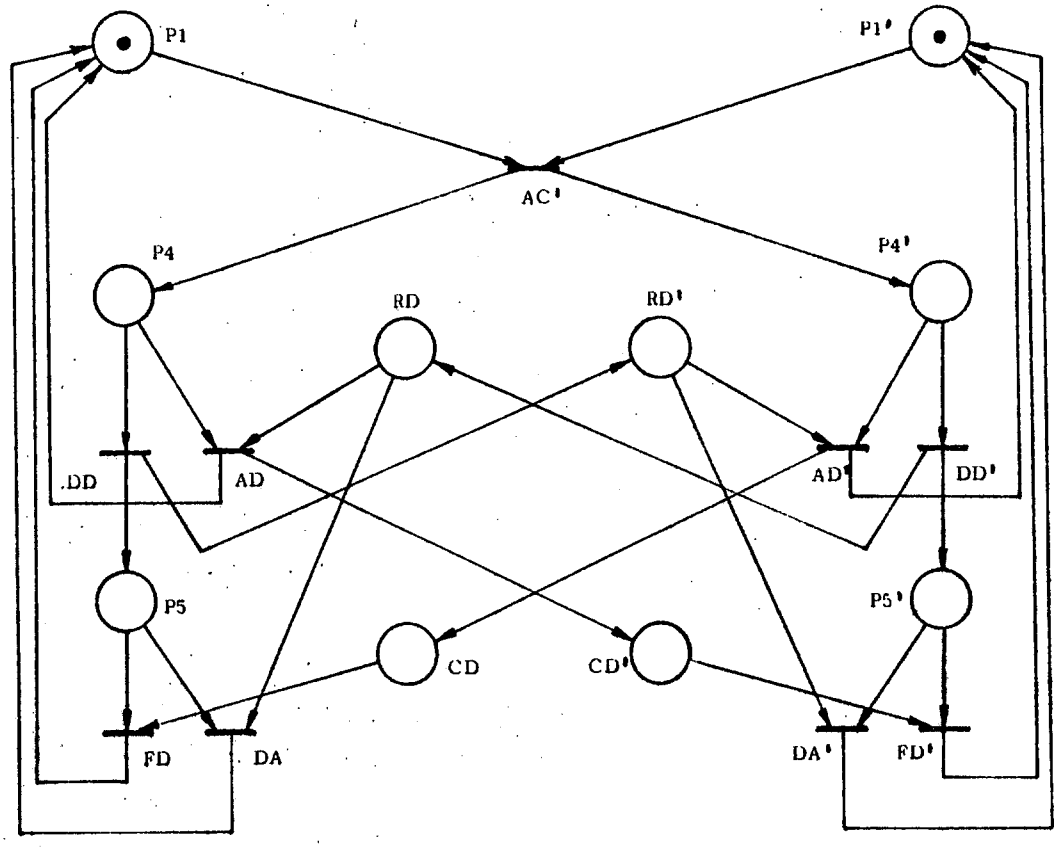


Figure 4. A result of applying the net reductions technique: improved ECMA protocol specification.

	AC'	DD	AD	FD	DA	DD'	AD'	FD'	DA'	I1	I2	I3	I4
P1	-1		1	1	1					1		1	
P4	1	-1	-1							1			1
P5		1		-1	-1					1			
RD			-1		-1	1						1	
CD				-1			1					1	
P1'	-1						1	1	1		1		1
P4'	1					-1	-1				1	1	
P5'						1		-1	-1		1		
RD'		1					-1		-1				1
CD'			1					-1					1

Table 2. The incidence matrix and invariants for the reduced net in Figure 4.

4. Research Directions

Berthelot and Terrat actually ran into some difficulty in modeling the other two phases of the ECMA protocol. They fabricated a solution to one of the problems by recourse to another form of the net called a predicate-transition net ((16)) which is capable of behaving as a fixed capacity queue which recognizes a finite set of message identifiers during the message transmission phase. The notation and analysis of predicate-transition nets is a little cumbersome. Jensen has done some work to bring this concept of the recognition of different tokens back within the fold of traditional net theory by formulating colored Petri net ((15)). This concept deserves more study in general and for its application to the present area of interest. Significantly, his paper includes an example of the design of a protocol for a distributed database system.

Also of interest is a further investigation of the matrix approach for the regular time nets of section 2. With the exception of Molloy, none of the authors mentioned there use a linear algebraic approach, and yet this is the single most useful analytic technique yet applied to Petri nets. One inherent problem in the canonical formulation of Petri nets which stands in the way of even greater usage of linear algebra is that the incidence matrix of the net is not square. The more powerful half of the matrix theory is concerned only with square matrices: the theory of determinants and eigenvectors. In so far as the eigenvectors of a matrix represent, in some sense, the optimal flows through the system, they could be of great interest for the present application area.

Finally, the promise of general net simulation and analysis systems needs to be realized. In this direction, more work needs to be done on the general category of Petri nets so that different net variants can also be defined. As our knowledge of tool and environment construction matures along with general net theory, this should become a realizable dream.

5. References

((1)) Petri: Communication with Automata, Information Systems Theory Project, technical report, 1965.

((2)) Peterson: Petri Net Theory and the Modeling of Systems, Prentice Hall, 1981.

((3)) Jennings and Hartel: Petri Net Simulations of Communication Networks, Naval Postgraduate School technical report, 1980.

((4)) Smart: Analytic Performance Modeling of Concurrent Computer Systems by Stochastic Petri Nets, Naval Postgraduate School techical report, 1981.

((5)) Sunshine: Formal Modeling of Communication Protocols, USC technical report, 1981.

((6)) Noe, Crowley, and Anderson: The Design of an Interactive Graphical Net Editor, University of Washington technical report, 1974.

((7)) Merlin: A Methodology for the Design and Implementation of Communication Protocols, IEEE Transactions on Communications, 24, 6. p.614-621, 1976.

((8)) Coolahan and Roussopoulos: Timing Requirements for Time-Driven Systems Using Augmented Petri Nets, IEEE Transactions on Software Engineering, 9, 5. p.603-616, 1983.

((9)) Molloy: Performance Analysis Using Stochastic Petri Nets, IEEE Transactions on Computers, 31, 9. p.913-917, 1982.

((10)) Genrich, Lautenbach, and Thiagarajan: Elements of General Net Theory, Lecture Notes in Computer Science, 84, Springer, 1980.

((11)) Molloy: Integration of Delay and Throughput Measures in Distributed Processing Models, UCLA technical report, 1982.

((12)) Ramamoorthy and Ho: Performance Analysis of Asynchronous Concurrent Systems Using Petri Nets, IEEE Transactions on Software Engineering, 4, 5. p.440-449, 1980.

((13)) Lipton: The Reachability Problem Requires Exponential Space, Yale University technical report, 1976.

((14)) Sifakis: Structural Properties of Petri Nets, Lecture Notes in Computer Science, 64, p.474-483, Springer, 1978.

((15)) Jensen: Colored Petri Nets and the Invariant Method, Theoretical Computer Science, 14, 3. p.317-336.

((16)) Berthelot, Terrat: Petri Net Theory for the Correctness of Protocols, IEEE Transactions on Communications, 30, 12. p.2497-2505.

((17)) Berthelot, Roucairol, and Valk: Reductions of Nets and Parallel Programs, Lecture Notes in Computer Science, 84, Springer, 1980.

OSNOVNA NAČELA DF SISTEMOV

J. ŠILC IN B. ROBIČ

INSTITUT „JOŽEF STEFAN"
LJUBLJANA

UDK: 681.519.7

Koncept krmiljenja s tokom podatkov predstavlja bistveno spremembo v načinu izvajanja instrukcij napram klasičnemu sekvenčnemu izvajanju. Ti sistemi ne delujejo na podlagi krmilnega toka, zato tudi ne potrebujejo programskega števca in pomnilnika s klasično funkcijo. V DF računalnikih postanejo instrukcije izvršljive v trenutku, ko prispe zadnji med zahtevanimi operandi, kar omogoča vzporedno izvrševanje instrukcij. Logična posledica tega je visoka stopnja izkoriščenosti inherentne paralelnosti, prisotne v algoritmih.

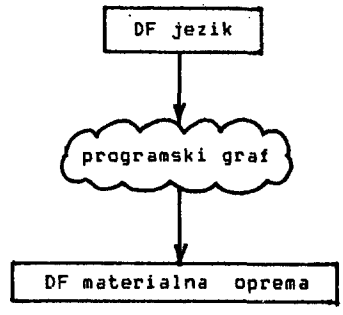
Basic Principles of Data Flow systems. The data flow concept is a fundamentally different way of looking at instruction execution in machine-level programs - an alternative to sequential instruction execution. In a data flow computer an instruction is ready for execution when its operands have arrived. There is no concept of control flow, and data flow computers do not have program location counters. A consequence of data - activated instruction execution is that many instructions of a data flow program may be available for execution at once. Thus, highly concurrent computation is a natural consequence of the data flow concept.

Uvod

Vse von Neumannove arhitekture računalniških sistemov imajo dve pomembni karakteristiki. Prvič, imajo globalni naslovljivi pomnilnik, ki pomni programe ter podatke in katerega vsebina se v skladu s programskimi instrukcijami pogosto ažurira. In drugič, vsebujejo programski števec, katerega vsebina je naslov naslednje instrukcije, ki naj se izvrši. Programski števec se bodisi implicitno ali eksplicitno ažurira in s tem določa sekvence instrukcij, ki se izvajajo. Takšno krmiljenje poteka programa je temeljna omejitev von Neumannovih arhitektur, še posebno pri vzporednem procesiranju.

DF računalniki (data flow computers) nimajo nobene od obeh zgoraj omenjenih značilnosti. Prvič, njihova struktura je zasnovana tako, da poteka procesiranje na osnovi vrednosti in ne naslovov spremenljivk, kar pomeni, da ni potreben globalni naslovljivi pomnilnik, saj ni nikakršnega naslavljanja. Drugič, v DF sistemih ni ničesar, kar bi bilo podobno programskemu števcu. Torej temeljijo ti sistemi na načelih asinhronosti in funkcionalnosti, ki pravita, da so vse operacije funkcije (funkcionalnost), ki postanejo izvršljive takoj, ko so na voljo vrednosti vseh vhodnih operandov (asinhronost). Iz načela funkcionalnosti sledi, da se katerikoli operaciji, ki sta izvršljivi, lahko izvršita po kakršnemkoli zaporedju ali konkurenčno.

Zgornji načeli logično vodita do programskega grafa (data flow program graph), v katerem točke ponazarjajo operacije, usmerjene povezave pa so nosilke vrednosti vhodnih operandov ter včasih še dodatnih informacij o delih izračunov, katerim pripadajo operandi. Programskemu grafu, ki je rezultat prevajanja programa v višjem DF jeziku (data flow language), se dinamično prilagodi materialna oprema (data flow hardware), ki je sposobna izvajanja vsega inherentnega paralelizma, opisanega s tem grafom , kar ponazarja slika 1 [1].



Slika 1.

* Raziskavo financira RSS po pogodbi C2-0124-106-84

Dinamična vzporednost

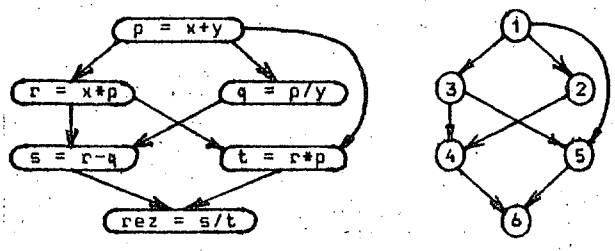
Veliko število algoritmov (npr. pri NP problemih) vsebuje inherentni paralelizem, ki pa v von Neumannovih arhitekturah ni izrabljen v največji možni meri. Arhitekture, ki bi to omogočale, bi npr. vse NP probleme prevedle na P probleme, kar pomeni, da bi se eksponentna časovna kompleksnost znižala na polinomsko. Zakaj ? Eksponentna časovna kompleksnost determinističnih algoritmov za reševanje nekaterih NP problemov izvira iz števila potencialnih rešitev, katere mora algoritem sekvenčno generirati in preveriti. S paralelnimi arhitekturami pa bi bilo moč realizirati nedeterministične algoritme, ki bi bili sposobni sočasnega generiranja potencialnih rešitev in izbire najustreznejše med njimi.

V ta namen je potrebno definirati programski jezik, ki bi dovolj eksplicitno izražal inherentno paralelnost algoritma, definirati strukturo (strojni jezik), ki je rezultat prevajanja programa v tem jeziku in ki omogoča dovolj enostavno realizacijo z materialno opremo; predvsem pa naj ohranja vso inherentno paralelnost algoritma.

Oglejmo si segment programa, zapisanega v enem od von Neumannovih jezikov.

```
1)  p := x+y ;
2)  q := p/y ;
3)  r := x*p ;
4)  s := r-q ;
5)  t := r*p ;
6)  rez := s/t ;
```

Standardni prevajalniki generirajo kodo, ki ustreza zaporednemu izvajanju stavkov segmenta, torej 1,2,3,4,5,6 . Vendar pa se lahko nekateri med njimi izvršijo paralelno, npr. 1,[2,3],4,5,6, kjer z [2,3] opišemo paralelno izvršitev stavkov 2 in 3. Vprašamo se, ali je to vsa inherentna paralelnost zgornjega segmenta. Očitno je, da je iz sekvenčnega zapisa paralelnost težko razvidna, zato se poslužujemo programskih grafov. Programski graf, ki ustreza zgornjemu segmentu, je prikazan na sliki 2.



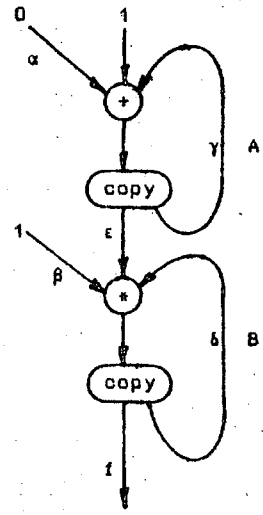
Slika 2.

Točke programskega grafa ponazarjajo stavke (operacije), usmerjene povezave pa sovpadajo s prehajanjem vrednosti operandov. Ker je graf na sliki 1 aciklični, se lahko operacije, ki so na istem nivoju, izvajajo paralelno. Operacija w je na nivoju i, če je dolžina najdaljše poti od začetne operacije do operacije w enaka i. V programskem grafu na sliki 1 so operacije porazdeljene po nivojih na sledeč način:

nivo 0 : 1
nivo 1 : 2,3
nivo 2 : 4,5
nivo 3 : 6.

Zaporedje izvrševanja stavkov 1,[2,3],[4,5],6 izkorišča inherentno paralelnost v največji meri. Operacije, ki so na istem nivoju, lahko kljub temu postanejo izvršljive v različnih časih - tedaj, ko so na voljo vrednosti vseh vhodnih operandov. Predpostavimo, da trajata operaciji seštevanja in odštevanja po eno, množenja dve in deljenja tri časovne enote. Čeprav sta operaciji 4 in 5 na istem nivoju, torej se lahko izvajata paralelno, bo postala operacija 4 izvršljiva po štirih, operacija 5 pa že po treh časovnih enotah. Torej je izvajanje programskega grafa popolnoma asinhrono.

Pri cikličnih grafih lahko nastanejo dodatne težave. Kadar v posamezni ponovitvi zanke naletimo na podatkovne odvisnosti med operacijami, to ne ustavi nadaljnih ponovitev, dasiravno predhodna ponovitev ni popolnoma končana, saj so vrednosti vhodnih operandov akumulirane v določenih vejah grafa. Oglejmo si primer intuitivno konstruiranega cikličnega programskega grafa za izračun funkcije f(i) = i! (slika 3).



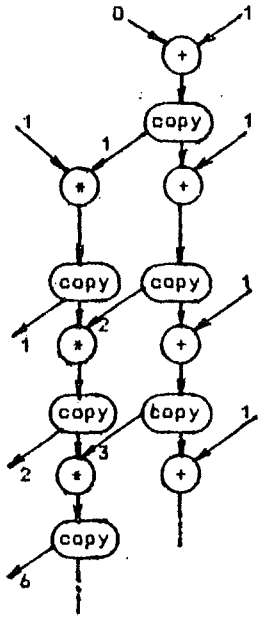
Slika 3.

Poseben selekcijski mehanizem poskrbi, da se ob prvi izvršitvi blokov A in B uporabita povezavi α in β ,ob vsaki naslednji pa namesto njiju povezavi γ in δ. Blok A inkrementira vrednost operanda za 1, s katerim nato blok B pomnoži prejšnji rezultat. Predpostavimo, da je za operacijo kopiranja (copy) potrebno pol, operacijo seštevanja ena in za operacijo množenja dve časovni enoti. Tedaj bi bile funkcijske vrednosti v točki f zaporedoma 1! , 2! , 4! ... Vzrok za izostanek rezultata 3! je v tem, da se zaradi hitrejšega izvajanja bloka A vrednost vhodnega operanda v povezavi ε prekrije z novo vrednostjo, ne da bi blok B uporabil predhodno vrednost. V primeru, ko bi se blok B izvajal hitreje kot blok A, pa bi se ista vrednost vhodnega operanda v povezavi ε uporabila večkrat, kar bi imelo za posledico, da bi se nekatere funkcijske vrednosti v točki f ponavljale, npr. 1!, 2! , 2! , 3! ...

Tako ni mogoče s prisotnostjo katerekoli vrednosti vhodnega operanda deklarirati vozlišča kot izvršljivega, saj lahko pripadajo vrednosti vhodnih operandov popolnoma različnim delom izračuna. Obstaja nekaj različnih rešitev nastalega problema oziroma načinov realizacije DF sistema [2].

(i) Ciklični programski graf transformiramo v aciklični tako, da vsako ponovitev opišemo z acikličnim podgrafom. Tako transformiran graf iz slike 3 ima obliko kot je prikazana na sliki 4.

Ta rešitev zahteva velike količine programskega pomnilnika, zahteva pa tudi dinamično generiranje koda v primeru, ko je pogoj za izstop iz zanke izračunan šele v času njenega izvajanja. Obe zahtevi se odražata kot pomembna pomankljivost v praktičnih sistemih.



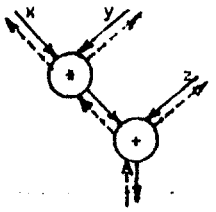
Slika 4.

(ii) Ponovitve opišemo z enim grafom, vendar se lahko ponovitev prične šele tedaj, ko je predhodna končana.

Takšna rešitev ne dovoljuje vzporednosti med ponovitvami in zahteva posebne ukaze ali posebno materialno opremo, ki testira konec ponovitve.

(iii) Uporaba grafov je omejena tako, da je v veji grafa prisotna istočasno samo ena vrednost operanda [3]. To pomeni, da se operacija lahko izvrši le tedaj, ko so prisotne vrednosti vseh vhodnih operandov in ni v izhodni veji nobene vrednosti. Torej vrednost spremenljivega operanda v neki povezavi ne sme biti spremenjena vse dokler ni uporabljena. Ko pa je enkrat uporabljena, mora postati neuporabljiva. Kljub sekvenčnosti izvajanja je prednost tega načina pred prvima dvema v možni izrabi cevljenja (pipelining). Vsaka operacija po svoji izvršitvi obvesti svoje "starše", da je pripravljena od njih sprejeti naslednje vrednosti vhodnih operandov. To stori s pomočjo dodatnih povezav - nosilk potrditvenih (aknowledge) signalov, kar kaže primer na sliki 5.

——— nosilka vrednosti operanda
– – – – – nosilka potrditvenega signala



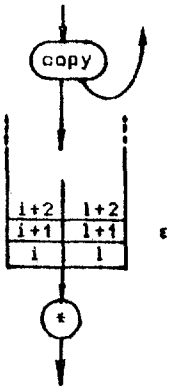
Slika 5.

Z uvedbo potrditvenih signalov se število povezav, in s tem promet v grafu, podvoji.

(iv) Poleg vrednosti operandov nosijo povezave še dodatno informacijo o delih izračunov, katerim pripadajo operandi [4]. Pravimo, da so povezave nosilke znakov (token). Znaki imajo dodatne labele, ki vsebujejo indeks in nivo ponovitve. Te labele običajno imenujemo barva. Operacije se lahko izvrši le tedaj, ko imajo vsi vhodni znaki enako barvo. Ta metoda uporablja popolnoma statično generiran kod in omogoča maksimalno vzporednost izvajanja.

Takšna rešitev zahteva povečan pretok informacij po grafu in dodatna vozlišča za spreminjanje ter primerjanje label, kar ima za posledico bodisi dodaten čas za izračun label ali pa uporabo posebne materialne opreme.

(v) Tudi tu so povezave nosilke znakov, poleg tega pa opravljajo še funkcijo vrst (queue), kar pomeni, da so v njih znaki razvrščeni v istem vrstnem redu, kot so vanje prihajali. Povezava ε s slike 3 ima tedaj obliko, kot jo prikazuje slika 6.



Slika 6.

Ta rešitev omogoča enako stopnjo vzporednosti kot pri labeliranju, zahteva pa čakalne vrste, ki so prostorsko zahtevne.

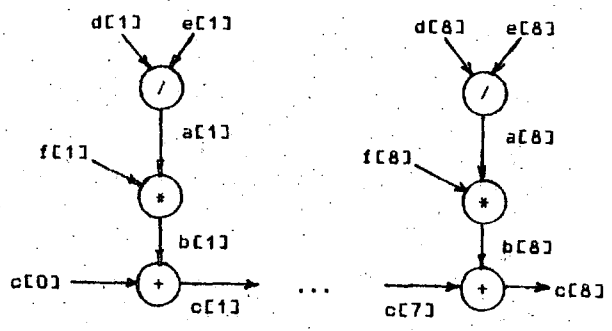
Programski graf, zgrajen po enem od zgornjih načel, je struktura, ki učinkovito povezuje visok programski jezik z materialno opremo in popolnoma ohranja inherentno paralelnost (slika 1).

Primerjavo opisanih petih realizacij DF sistema si oglejmo na primeru naslednjega programa:

```
input d,e,f
   c[0] = 0
for i from 1 to 8 do
   begin
      a[i] = d[i] / e[i]
      b[i] = a[i] * f[i]
      c[i] = b[i] + c[i-1]
   end
output a,b,c
```

Predpostavimo, da zahteva deljenje tri, množenje dve in seštevanje eno časovno enoto. Namišljeni DF računalnik naj ima štiri procesne enote P1, P2, P3 in P4, od katerih lahko vsaka izvaja katerokoli od operacij. Idealizirajmo DF računalnik tako, da bodo zakasnitve pomnilnika in med povezavami nič. Program se bo izvrševal po programskem grafu, prikazanem na sliki 7.

Iz programskega grafa na sliki 7, je razvidno, da je kritična pot a[1], b[1], c[1], c[2], ... , c[8], katere minimalni čas izvajanja je 13 časovnih enot.



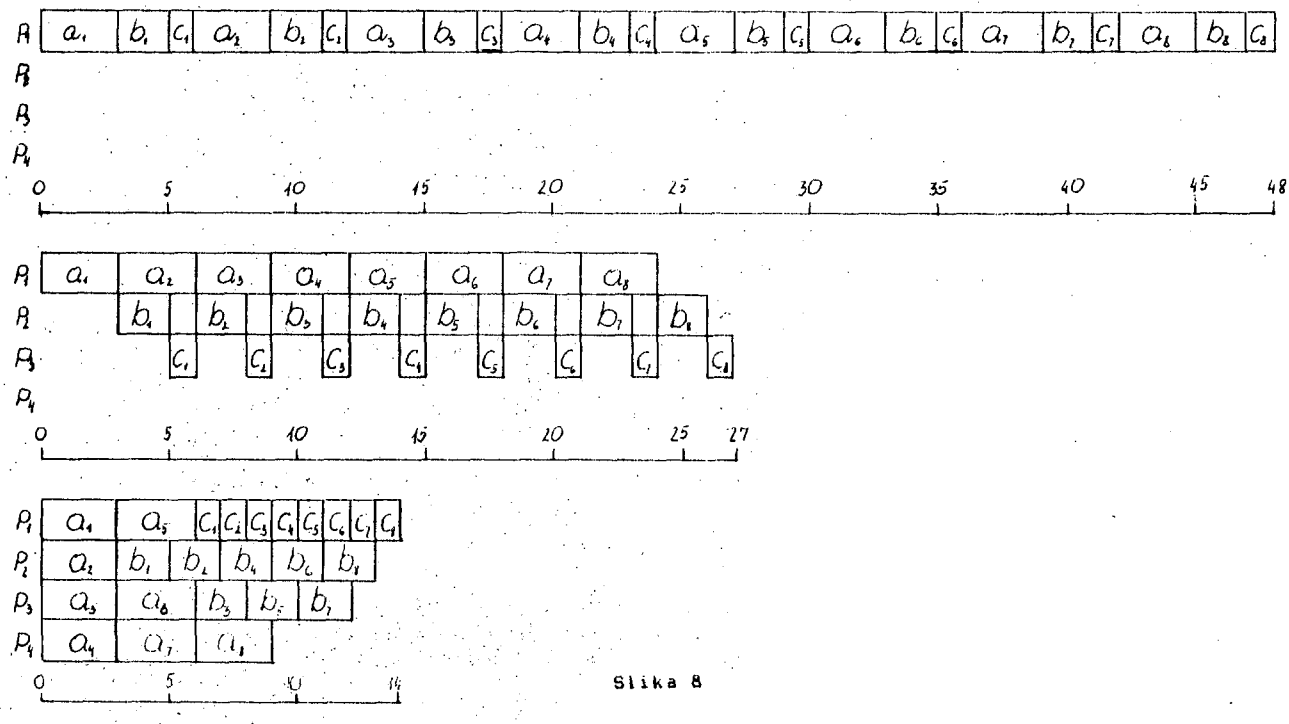
Slika 7.

Če uporabimo strategijo (ii), po kateri se naslednja ponovitev prične šele ko se predhodnja konča, dobimo popolnoma sekvenčno izvajanje programa. Ker zahteva vsaka ponovitev po 6 časovnih enot, je skupni čas izvajanje programa enak 6 X 8 = 48 časovnih enot. Kot rečeno, je izvajanje programa sekvenčno, torej bi zadoščal en procesor; vendar je praktično izračun porazdeljen preko vseh štirih procesorjev, tako da postane uporabljenost procesorjev 12 / 48 = 0.25 (slika 8.a). Strategija (iii), ki dovoljuje istočasno prisotnost enega znaka v veji programskega grafa, vodi do cevljenja bloka prireditvenih stavkov znotraj zanke (slika 8.b). Čas izvajanja zanke narekuje časovno najzahtevnejša operacija v zanki (deljenje), tako je čas izvajanja programa enak 3 X 8 + 3 = 27 in uporabljenost procesorjev 12 / 27 = 0.44. Strategije (i), (iv) in (v) so povsem enakovredne in omogočajo najhitrejše izvajanje programa in sicer 14 časovnih enot, pri najboljši uporabljenosti procesorjev 12 / 14 = 0.86 (slika 8.c).

Iz primera vidimo, da imajo sekvenčni sistemi najslabše in DF sistemi z labeliranimi znaki najboljše performance, medtem ko so sistemi, ki uporabljajo koncept cevljenja nekje vmes.

DF jeziki

Osnovni cilj vseh izboljšav klasične von Neumannove arhitekture je v čimvečji izrabi paralelnosti. Hkrati z izboljšavami v arhitekturi so potekale raziskave tudi na področju optimizacije prevajalnikov, ki so programe, pisane v konvencionalnih von Neumannovih jezikih, prevedli v obliko, prirejeno izboljšani arhitekturi. Poleg tega pa je bilo konstruiranih tudi nekaj novih visokih programskih jezikov, kot sta Concurrent Pascal in Glypnir, prirejenih izboljšanim von Neumannovim arhitekturam. Ti jeziki vsebujejo sintaksne konstrukte, s pomočjo katerih postanejo arhitekturne lastnosti računalnika vidne za progra-



Slika 8

merja. Le-ta z njihovo uporabo olajša delo prevajalniku pri odkrivanju paralelnosti. Večina teh jezikov pa je nenaravnih v smislu, da v preveliki meri odražajo arhitekturo, na podlagi katere so oblikovani, manj pa način, na katerega programer razmišlja pri reševanju nekega problema.

Tako kot ostale oblike paralelnih računalnikov zahtevajo tudi DF računalniki (zaradi čimboljše izrabe svojih lastnosti) posebne visoke programske jezike, t.i. DF jezike, ki pa ne sodijo med von Neumannove jezike [5].

Za razliko od konvencionalnih jezikov, kateri delujejo nad podatki s pomočjo stranskih učinkov, DF jeziki le-teh ne poznajo. Znana je namreč zveza med učinkovitim paralelnim računanjem in odsotnostjo stranskih učinkov. To lastnost imajo funkcionalni jeziki, ki delujejo izključno na podlagi uporab funkcij nad vrednostmi [6]. Sledeča lastnost, katero imajo DF jeziki, je lokalnost učinka, kar pomeni, da ukazi nimajo nepotrebnih, daleč segajočih podatkovnih odvisnosti. Omejitve glede izvajanja ukazov temeljijo izključno na podatkovnih odvisnostih. Posledica te zahteve je, da je vsa informacija, potrebna za izvajanje programa, vsebovana v programskem grafu.

DF jeziki imajo v sintaksnem smislu veliko skupnih lastnosti s konvencialnimi jeziki, saj uporabljajo podobne programske konstrukte, kot so prireditev, aritmetične izraze, pogojne stavke, iteracije in rekurzijo. Bistveno pa se razlikujejo v semantiki. Za razliko od konvencionalnih jezikov, pri katerih identifikator predstavlja naslovljivo enoto pomnilnika, pa pri DF jezikih predstavlja povezavo. Posledica takšne semantike DF jezikov je, da se lahko nekemu identifikatorju priredi vrednost samo enkrat, kar imenujemo pravilo o enkratni prireditvi. S preimenovanjem identifikatorjev lahko v neiterativnih delih programa problem večkratne prireditve uspešno rešimo. Težave pa lahko nastopijo v zankah. Zato je potrebno pri definiranju zanke navesti (1) vhodne vrednosti vseh začetnih identifikatorjev, (2) pogoj ustavljanja, (3) vrednost, ki naj bo rezultat ob koncu izvajanja zanke in (4) pravila po katerih se začetnim identifikatorjem (ob vsaki izvršitvi telesa zanke) priredijo nove vrednosti.

Za ilustracijo si oglejmo primer algoritma za izračun n!, zapisanega v DF jezikih VAL (Value-oriented Algorithmic Language)

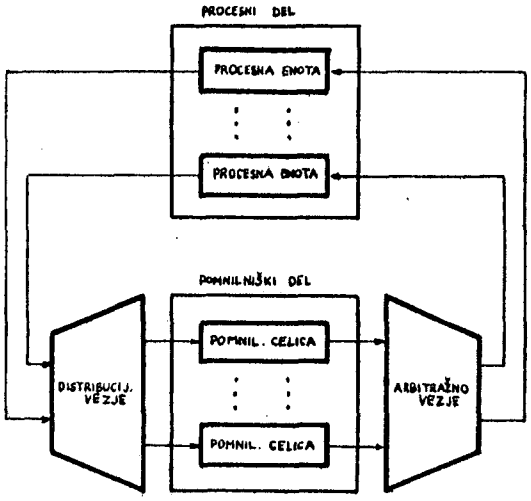
```
for j,k := n,1 ;                          (1)
do if j = 0                               (2)
     then k                               (3)
     else iter j,k := j-1,k*j ;           (4)
   end
end
```

in ID [5]

```
(initial j ← n; k ← 1                     (1)
 while j <> 0 do                          (2)
   new j ← j-1; new k ← k*j;              (4)
 return k)                                (3)
```

Arhitektura DF sistema

DF računalniki nimajo centralnega procesorja, temveč imajo procesni del, ki ga sestavlja množica nekaj deset, sto ali tisoč procesnih enot. Vsaka procesna enota je enaka enostavni aritmetično logični enoti ali vhodno/izhodnemu procesorju. Nimajo niti pomnilnika, kakršnega uporabljajo von Neumannove arhitekture, zato pa imajo pomnilniški del, ki ima potencialno veliko število pomnilniških celic, v katerih se nehajajo vsi podatki o programskem grafu. Za DF računalnike je značilno tudi to, da ne uporabljajo sinhronizacijske ure, programskega števca in registrov. Zato pa ima arbitražno vezje, ki usmerja izhode iz celic pomnilniškega dela v ustrezne procesne enote procesnega dela in distribucijsko vezje, ki povezuje procesne enote s pomnilniškimi celicami. Arhitektura DF sistema je prikazana na sliki 9 [7].



Slika 9.

Arbitražno vezje ugotovi, katere operacije so godne za izvršitev in jih posreduje procesnemu delu. Le-ta jih izvrši in pošlje delne rezultate distribucijskemu vezju. To vezje pa ugotovi katere operacije iz pomnilniškega dela čakajo na dobljene rezultate in jim jih posreduje v obliki vhodnih operandov. Sedaj zopet nastopi arbitražno vezje s čimer se cikel ponovi.

Zaključek

Osnovne ideje, na katerih temelji DF koncept, segajo v pozna šestdeseta leta. Z razvojem sodobne VLSI tehnologije je postala obstoječa (von Neumannova) arhitektura glavna ovira pri izkoriščanju paralelnosti v algoritmih. VLSI tehnologija pa je omogočila smiselno uporabo materialne opreme v arhitekturah non-von (ne von Neumannovih) sistemov. DF koncept šestdesetih let je tako postal uresničljiv, saj omogoča ta tehnologija izdelavo integriranih vezij s celo 100 nožicami. Tisoč takih vezij, ki opravljajo usmerjevalno-povezovalno funkcijo (router), omogoča uporabo celo 512 procesnih enot ali celičnih blokov (cell block). Če vsaka od procesnih enot izvrši milijon instrukcij v sekundi, potem nove arhitekture (ob pravilni izrabi paralelizmov) omogočajo skoraj milijardo inštrukcij v sekundi.

Večji DF projekti v svetu potekajo :

- na MIT, kjer skupina pod vodstvom J.Dennisa razvija DF sistem z rezinastimi procesorji tipa Am2903,
- tudi na MIT, kjer skupina pod Arvindovim vodstvom gradi VLSI 64 procesorski DF računalnik z labeliranimi znaki (tagged-token),
- na univerzi Utah deluje skupina pod vodstvom A.Davisa, ki je sestavil prvi delujoči DF računalnik,

- na CERT v Toulousu, kjer deluje skupina pod vodstvom J.C.Syra na projektu LAU,
- na univerzi v Manchesteru gradijo eksperimentalni multiprocesorki DF sistem z labeliranimi znaki pod vodstvom J.Gurda in I. Watsona in
- na univerzi Tokyo, računalnik Topstar, pod vodstvom T.Suzukija in J.Motooke.

Analize učinkovitosti, ki so jih izvršili na realiziranih DF sistemih, so pokazale občutno časovno izboljšanje pri reševanju nekaterih znanih algoritmov, kot sta FFT (40:1) in Gaussova eliminacijska metoda (80:1). Ker so ti sistemi šele v razvoju, obstaja tudi nekaj nerešenih problemov, kot sta problema vhodno/izhodnih aktivnosti in začasnega pomnenja [2].

Literatura

[1] T.Agerwala, Arvind : 'Data Flow Systems', Computer, Vol.15, No.2,Feb.1982, pp.10-13

[2] D.D.Gajski, D.A.Padua, D.J.Kuck & R.H. Kuhn : 'A Second Opinion on Data Flow Machines and Languages', Computer,Vol.15, No.2, Feb.1982, pp.58-69

[3] J.B.Dennis : 'Data Flow Supercomputers', Computer,Vol.13,No.11,Nov.1980, pp.48-56

[4] I.Watson, J.Gurd : 'A Practical Data Flow Computer', Computer,Vol.15,No.2,Feb.1982, pp.51-57

[5] W.B.Ackerman : 'Data Flow Languages', Computer,Vol.15,No.2,Feb.1982, pp.15-25

[6] J.Backus : 'Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs', Comm. of the ACM, Vol.21,No.5,Aug.1978, pp.613-641

[7] J.B.Dennis, W.Y.-P.Lim & W.B.Ackerman : 'The MIT Data Flow Engineering Model', Information Processing 83, R.E.A.Mason (ed.), Elsevier Science Publishers B.V. (North-Holland), pp.553-560

INFORMATICA 2/85

O JEZICIMA ZA KONKURENTNO PROGRAMIRANJE KAO SREDSTVU ZA PROJEKTOVANJE UPRAVLJAČKIH SISTEMA

GOJO ŠTRKIĆ,
DAVORIN NOVOSEL

ELEKTROTEHNIČKI FAKULTET
BANJALUKA

UDK: 681.326.7

SADRŽAJ - U radu je dat osvrt na jezike za konkurentno programiranje kao sredstvo za projektovanje manjih upravljačkih sistema implementiranih na digitalnim procesorima (npr. mikroprocesorima). U analizu je uzet konkurentni paskal kao prvi takav jezik i jezici modula i edison kao manje glomazni jezici podesniji za manje sisteme.

ABSTRACT - This paper gives analysis of concurrent programming languages as the tool for design of less complex control systems implemented on digital processors (e.g. microprocessors). The analysis treats concurrent pascal as such first language and modula and edison as smaller languages more convenient for less complex systems.

1. UVOD

Upravljački sistem implementiran na digitalnom procesoru (npr. mikroprocesoru) se može shvatiti kao skup labavo povezanih procesa (zadataka) od kojih je svaki odgovoran za po jednu funkciju ili grupu odgovarajućih upravljačkih funkcija. Da bi opsluživao i upravljao sa više periferala upravljački sistem mora biti u stanju da istovremeno izvodi više procesa. Sa jednim procesorom prava je paralelnost, naravno, nemoguća pošto procesor može u jednom trenutku da izvodi samo jednu instrukciju. Zbog toga se zahtijevaju neki mehanizmi za alociranje procesorskog vremena za potrebe procesa kao i za upravljanje i sinhronizaciju istih. Pored toga upravljački sistem može imati zahtjeve da se opslužuje asinhroni ulaz/izlaz i da se odgovara na dogadjaje koji su vremenski uslovljeni (zahtjevi realnog vremena).

Upravljačka logika može biti ugradjena u sam program i isti direktno implementiran na procesor. Medjutim, kod kompleksnijih sistema (koji sadrže veći broj procesa) najbolje je upravljačke funkcije ugraditi u poseban dio, tzv. kernel sistema, koji se može projektovati pomoću jezika za konkurentno programiranje |9|.

2. UOPŠTENO O KONKURENTNIM JEZICIMA

Razvoj jezika za konkurentno programiranje je još uvijek u eksperimentalnoj fazi iako je relativno mnogo vremena prošlo kad se pojavio konkurentni paskal kao prvi jezik za konkurentno programiranje |4|.

Začeci konkurentnog programiranja datiraju iz vremena kada je program izdijeljen i sekvencijalne cjeline nazvane procesima koje su se mogle izvoditi asinhrono. Ako je svaki proces operirao na svojim sopstvenim podacima onda se uz pomoć hardverskih mehanizama zaštite sistem od više procesa ponašao na sasvim predvidiv način. Medjutim, problemi su nastali kada je zatrebalo da procesi saradjuju oko dijeljenja nekih zajedničkih nedjeljivih resursa ili kada je trebalo da saradjuju na nekom zajedničkom zadatku.
Za razumijevanje ovih problema pronadjeni su koncepti kritičnih regiona i semafora |7|.

Dijkstra |1| je projektovao jedan multiprogramski sistem koji se može shvatiti kao prvi primjer kako treba sistematski organizovati softver. Njegov multiprogramski sistem je organizovan kao jedna cjelina sastavljena od nekoliko hijerarhijski postavljenih slojeva koji fizičku mašinu pretvaraju u jednu virtualnu apstraktnu mašinu koja pruža značajne prednosti kako prilikom upotrebe tako i prilikom projektovanja iste. Ove osnovne ideje ugradjivane su kasnije u gotovo sve jezike za konkurentno programiranje.

Osnova na kojoj su jezici za konkurentno programiranje projektovani je da se išlo na odabiranje minimalnog broja osnovnih struktura koje bi bile dovoljne da programer pomoću njih može jednostavno i pouzdano graditi svoje programe. Kod svih programskih jezika za konkurentno programiranje mogu se naći tri zajedničke osobine a to su:

1. mogućnost za predstavljanje konkurentnih aktivnosti (procesi, zadaci),

2. mogućnost za ostvarivanje komunikacije i sinhronizacije izmedju tih konkurentnih aktivnosti,

3. mogućnost za obavljanje U/I aktivnosti u kojima su odredjeni problemi koji se javljaju sakriveni od programera korisnika.

Svi jezici za konkurentno programiranje baziraju se na jednom rezidentnom programu često zvanom kernel ili nukleus datog jezika. Kernel jezika za konkurentno programiranje podržava implementaciju procesa, sinhronizacije i U/I aktivnosti. Dakle, on obavlja veoma bitne operacije upravljačkog sistema te na taj način odredjuje propusnost, efikasnost i pouzdanost upravljačkog sistema projektovanog pomoću datog jezika.
Prema tome analiza multiprogramskog dijela jednog jezika za konkurentno programiranje veoma je bitna za analizu multiprogramskog dijela upravljačkog sistema.

3. KOMPARATIVNA ANALIZA JEZIKA: KONKURENTNI PASKAL, MODULA I EDISON

Za analizu ćemo odabrati nekoliko jezika za konkurentno programiranje i razmatraćemo njihov profil prema potrebama projektovanja upravljačkih sistema implementiranih na digitalnim procesorima. Osvrnućemo se na konkurentni paskal, jezik modula |8| i jezik edison |6| jer svaki od njih sa sobom nosi neke nove koncepcijske postavke bilo da su te postavke rezultirale iz iskustva nastalog na osnovu poznavanja prethodnih jezika za konkurentno i sistemsko programiranje bilo da su one uzrokovane tehnološkim napredkom koji je omogućio implementaciju odredjenih poznatih koncepata.

Nesumnjivo je da je pronalazak apstraktnog tipa podataka bio osnova na kojoj su se bazirali viši programski jezici za sekvencijalno i konkurentno programiranje. Naime, došlo se do mogućnosti da se pomoću odredjenih sintaktičkih struktura opišu odredjeni podaci kao i operacije koje je moguće obaviti nad istim a da se to može izdvojiti u jednu nezavisnu cjelinu koju je moguće nezavisno projektovati i testirati. Te strukture pomoću kojih se opisuju apstraktni tipovi podataka su različite u različitim programskim jezicima kako sekvencijalnim tako i u konkurentnim. Kako mi ovdje vršimo osvrt na jezike za konkurentno programiranje to ćemo ovdje spomenuti jedino apstraktne strukture karakteristične za ove jezike. To su monitori, procesi i klase u konkurentnom paskalu a moduli u jezicima modula i edison.
Ako se kritički osvrnemo na konkurentni paskal može se reći da je njegov koncept jedna dosta kompleksna tvorevina sastavljena od zajedničkih varijabli, procedura, rasporedjivanja procesa i modularnosti. Poznato je da monitor kao jedan od temelja konkurentnog paskala apstraktni tip podataka na kom se sve operacije obavljaju pomoću procedura koje pozivaju konkurentni procesi. U slučaju da više procesa istovremeno pozove monitorsku proceduru oni će stati u listu čekanja i izvoditi se striktno jedan iza drugoga.

Iako se koncept monitora jezika modula nešto razlikuje od koncepta monitora konkurentnog paskala on je u suštini isti s tim da su imena monitor i varijabla čekanja (queue) zamijenjeni sa imenima vezni modul (interface module) i signal. Kod jezika edison pošlo se od jednog praktičnog iskustva da je kod većine dobro izvedenih sistema takav slučaj da svaki proces potroši veći dio svog vremena na lokalnim podacima a manji dio vremena na izmjeni podataka sa drugim procesima i sinhronizaciji sa istim. Kad se tome doda činjenica da je tehnološki napredak omogućio implementaciju kritičnog regiona kog je mnogo ranije predložio Hoare |7| onda nije slučajno što je u edisonu izostavljen koncept monitora. Monitor se u edisonu gradi programiranjem pomoću jednostavnijih koncepata kao što su varijable, moduli, procedure i WHEN iskazi.

Prije pojave jezika edison u jezicima za konkurentno programiranje bila su poznata dva osnovna pristupa u implementaciji procesa. Jedan pristup je da se svaki proces nezavisno može pojaviti i nestati |3| a drugi je da proces poslije kreiranja stalno egzistira |5|. Pošto je i jedan i drugi pristup imao svoje nedostatke onda se pokušalo napraviti treći pristup u koji bi se ugradili elementi iz prethodna dva pristupa koji su se dobro pokazali u praksi. Na taj način su procesi u edisonu kao novijem jeziku za konkurentno programiranje prikazani pomoću konkurentnih iskaza

```
cobegin 1 do SL1
   also 2 do SL2
   ...
   also n do SLn end
```

Kod ovog pristupa procesi se mogu dinamički pojaviti i nestati ali svi istovremeno. (Brojevi u gornjim iskazima služe zato da se svaki proces može dodijeliti posebnom procesoru ili pak da se npr. memorijski prostor može podjednako podijeliti medju procesima. Sa SL je označena

lista iskaza).

Za razliku od edisona kod jezika konkurentnog paskala i modula procesi se opisuju pomoću specijalne vrste modula. Ovi specijalni moduli kao sintaktički oblici omogućavaju kompajlerima da obezbijede da jedan proces ne referencira varijable drugog procesa kako bi se izbjegle vrlo nezgodne vrste programskih grešaka.

U edisonu se išlo na gornji pristup konceptu procesa polazeći od pretpostavke da se ista pouzdanost može postići pomoću adaptiranja programskog stila u kom se procesi opisiju pomoću procedura koje se pozivaju pomoću konkurentnih iskaza. Pouzdanost ovakvog rješenja obezbijedjena je na taj način što je osigurano da svaki poziv procedure kreira nove lokalne varijable na koje jedino može da pristupi proces koji datu proceduru poziva.

Što se tiče tretiranja ulazno/izlaznih operacija evidentna je izvjesna evolucija. Naime, u konkurentnom paskalu je za ulazno/izlazne operacije postojala standardna procedura io ugradjena u kernel jezika i pomoću nje su se obavljale ulazno/izlazne operacije. Pomoću poziva procedure io (podaci, operacije, periferal) proces je obavljao jednu nedjeljivu operaciju koja daje ponovljive rezultate a za to vrijeme ostali resursi sistema su drugim procesima na raspolaganju.
Ovaj pristup ima nedostatke pošto je za svaki novi periferal koji se uvodi u sistem projektovan na bazi konkurentnog paskala potrebno modificirati kernel konkurentnog paskala što je dosta ozbiljan posao.

Uočavajući ovaj problem projektanti sljedećeg jezika za konkurentno programiranje su imali takav pristup da su obezbijedili takav mehanizam (moduli periferala - device modules) pomoću kog programer korisničkih programa može sam da piše manipulatore periferalima (device drivers). Za razliku od konkurentnog paskala i modula-e u kojima su prekidi tretirani na nivou mašinskog jezika u edisonu kao novijem jeziku za konkurentno programiranje prekidi su ignorisani čak i na nivou mašinskog jezika iz razloga što je implementacija namijenjena mikroprocesorima koji su jeftine komponente.

Pošto edison ignoriše prekide to se kod njega procesi jedino rasporedjuju po cikličkom redosljedu. Medjutim, zbog jednostavnosti rasporedjivanja vrijeme prebacivanja s jednog procesa na drugi je značajno krače u poredjenju sa sistemima projektovanim u konkurentnom paskalu iako je rasporedjivanje u konkurentnom paskalu na bazi prekida.

Ako se napravi jedan presjek kroz edison kao jezik baziran na bazi iskustva od nekoliko programskih jezika za konkurentno programiranje kao i na bazi sadašnjih tehnoloških mogućnosti da se uočiti da je povećana fleksibilnost jezika na račun pouzdanosti istog. Naime, išlo se od pretpostavke da jezik ne treba opterećivati nekim kompleksnim glomaznim apstrakcijama već da treba odabrati jedan manji skup jednostavnijih koncepata pomoću kojih se jednostavno mogu graditi željene kompleksnije strukture. Jasno se vidi da se na taj način žele izbjeći visoko specijalizovani i glomazni jezici čiji su nedostaci jedne prirode a istovremeno se želi izdići iznad asemblerskih jezika čiji su problemi druge prirode.

4. ZAKLJUČAK

Bitno je napomenuti da u projektovanju konkurentnih programskih jezika, koji se koriste za projektovanje raznih upravljačkih sistema, dolazi do izvjesnog zaokreta. Napuštaju se pristupi koji su apstraktne strukture gradili po uzoru na klasu jezika simula |2| a favorizuju se pristupi u kojima su jezici s jedne strane rasterećeni nepotrebnih glomaznih jezičkih koncepata a s druge strane u kojima su ugradjene odredjene jezičke primitive čije je ugradnje omogućio sadašnji tehnološki napredak. Trend je takav da se prekid kao takav ignoriše, i da se ide na gradnju takvih jezika koji podržavaju multiprocesorske konfiguracije.

LITERATURA

1. Dijkstra E.W. "Cooperating sequential processes", Programming Languages, F.Genuys, Ed., Academic Press, New York 1968.
2. Dahl, O.J., Dijkstra, E.W., and Hoare, C.A. R., "Structured programming", Academc Press, New York, NY, 1972.
3. Hansen P.B. "The RC 4000 real-time control system at Pulawy" BII7, BIT6, 1,pp.1-23,1966.
4. Hansen P.B.: "The programming language Concurrent Pascal", IEEE Transactions on Software Engineering 1,2 pp 199-207, June 1975.
5. Hansen P.B.: "Architecture of the Concurrent programs", Prentice Hall 1977.
6. Hansen P.B.: "Edison - a Multiprocessor Language", Software Practice & experience, 11, 4 (April 1981).
7. Hoare, C.A.R., "Monitors: An operationg system structuring concept" Comm. ACM 17, 10. (Oct. 1974), 549-557.
8. Wirth N.: "Modula: A language for multiprogramming", Software Practice & Experience, 7.1 (Jan. 1977.) pp (3-35).
9. Ž.Salčić, G.Štrkić: Projektovanje mikroprocesorskih izvršnih sistema za rad u realnom vremenu pomoću jezika za konkurentno programiranje, Automatika 1982, No 3-4.

PROŠIRENJE LISP KIT LISP JEZIKA FUNKCIJAMA EXPLODE I IMPLODE

NENAD MITIĆ
VOJISLAV STOJKOVIĆ

UDK: 519.682 LISP

REPUBLIČKI ZAVOD ZA STATISTIKU
SR SRBIJE, BEOGRAD
INSTITUT ZA MATEMATIKU, BEOGRAD

U radu je izloženo proširenje LispKit Lisp jezika [1] funkcijama EXPLODE i IMPLODE [2],[3]. Navedene funkcije se ne mogu definisati kao korisničke funkcije. Originalna programska implementacija je izvršena modifikacijom:
- prevodioca LispKit Lisp jezika na mašinski jezik SECD mašine i
- proširenjem simulatora SECD mašine.

Rad se završava navođenjem nekih primena navedenih funkcija.

An extension of LispKit Lisp language by EXPLODE and IMPLODE functions: implementation and application. An extension of LispKit Lisp language [1]by EXPLODE and IMPLODE functions [2],[3] is described in this work. The mentioned can not be defined as user's function.The orisinal program implementation is done by:
- a modification of the compiler of LispKit Lisp language into language of SECD machine, and
- an extension of the simulator of SECD machine.

The work is finished by giving some applications of mentioned functions.

1. Funkcionalni jezici

Poslednjih nekoliko godina smo svedoci naglog porasta interesovanja za funkcionalne (aplikativne) jezike. Razvoj programskih jezika baziran na strukturnom programiranju nije dao očekivane rezultate. Razlog relativnog neuspeha strukturnog programiranja leži u tome što nije došlo do potpunog raskida sa principima na kojima počivaju konvencionalni programski jezici (FORTRAN i COBOL).

Imperativni jezici, kako oni stariјeg tako i oni novijeg datuma(PASCAL,ADA), imaju osobinu da ne mogu na relativno jednostavan način da opišu postojeće (poznate) matematičke strukture i obratno, poseduju strukture podataka i naredbe koje se ne javljaju u matematici. Tako na primer ne postoji matematički ekvivalent poziva procedure (funkcije) koji daje različite vrednosti za iste vrednosti parametara. U matematici funkcija je uređen par (x,F(x)) gde se svaki put za isto F i isto x dobija ista vrednost funkcije F(x). U matematici promenljive ne mogu da menjaju vrednost, dok se izraz koristi samo da označi vrednost i u istom kontekstu isti izrazi imaju istu vrednost.

Nedostaci imperativnih programskih jezika proizlaze iz samog njihovog koncepta. Imperativni programi se izvršavaju na računarima Von Neumannovog tipa. Von Neumann je 1940 godine dao model računara koji se sastoji iz dva osnovna dela: pasivne memorije i aktivnog procesora.

U čisto funkcionalnim programskim jezicima navedeni nedostaci su otklonjeni. Funkcije se karakterišu isključivo svojim vrednostima. Ne postoji bočni efekat a samim tim nema ni menjanja vrednosti promenljivih. Neposredno se realizuju funkcije višeg reda (tj. funkcije čiji su argumenti druge funkcije) kao i komponovanje funkcija (vrednost rezultata zavisi isključivo od vrednosti argumenata). Program je funkcija čija je vrednost rezultat izvršavanja programa.

2. SECD mašina i EXECUTE funkcija

SECD mašina je model računara ne Von Neumannovog tipa. Ime SECD potiče od imena četiri glavna registra:

- S - stack. Sadrži međurezultate izračunavanja vrednosti funkcije(programa)
- E - environment. Sadrži vrednosti dodeljene promenljivim za vreme izračunavanja vrednosti funkcije
- C - control. Koristi se za čuvanje funkcija čija se vrednost izračunava, tj. programa na mašinskom jeziku koji se izvršava
- D - dump. Koristi se kao stek (magazinska memorija) za čuvanje vrednosti ostalih registara pri pozivu (računanju vrednosti) nove funkcije.

Program napisan na LispKit Lisp jeziku ili na mašinskom jeziku SECD mašine je valjani S-izraz tj. S-izraz koji ima vrednost. Ulazni i izlazni podaci programa su S-izrazi.

SECD mašina se može zadati kao funkcija EXEC. Definišimo funkciju EXEC na sledeći način:

$$EXEC(f^*,a)=apply(f,a)$$

Argumenti funkcije EXEC su:
- f^*, prevod funkcije f na mašinski jezik SECD mašine i
- a , argument funkcije f.

Vrednost funkcije EXEC je S-izraz čija je vrednost rezultat primene funkcije f na argument a.

Funkcija EXEC, tj. programska implementacija SECD mašine, može se definisati na sledeći način(na jednom jeziku algololikog tipa):

```
exec ; proc(fn,args)
begin
     (* vrše se inicijalizacije *)
     cycle ; case ivalue(car(c)) of
             begin
              1 ; ...
              2 ; ...
                  .
                  .
                  .
             21 ; goto endcycle;
                  .
                  .
                  .
             end;
             goto cycle;
     endcycle ; return car(s)
end;
```

Svakoj mašinskoj naredbi odgovara kôd koji označava blok naredbi koje se izvršavaju kad je ta naredba u pitanju.

Funkcija exec je osnovna funkcija programskog simulatora SECD mašine.Parametri funkcije exec su:

- fn: pokazivač prevoda funkcije f na mašinski jezik;
- args: pokazivač liste argumenata funkcije zadate u obliku S-izraza.

Vrednost funkcije exec je pokazivač prvog elementa steka S (rezultat izračunavanja). Prevodilac automatski dodaje dve naredbe : AP (apply) - mašinski kôd 4 i STOP - mašinski kôd 21.

2.1 Specifičnosti programske implementacije

Implementacija je izvršena na programskom jeziku PL/1 računara IBM 370/3031 Republičkog zavoda za statistiku SR Srbije u Beogradu. Implementacija je pojednostavljena zahvaljujući sledećim osobinama PL/1 jezika:

- jednostavno manipulisanje raznim tipovima podataka
- mogućnost redefinisanja promenljivih. Na taj način se lako dobijaju cifre broja koji je character tipa;
- korišćenje promenljivih promenljive dužine
- mogućnost konkatenacije podataka character tipa i brojeva u PICTURE formatu.

Neki detalji implementacije:

- celi brojevi su oblika binary fixed(31,0) (na dužini 32 bita). Smešteni su u nizu ivalue koji je istog tipa kao i brojevi
- character promenljive su maksimalne dužine 15 (tip character(15) varying) i smeštene su redom u niz stringstore istog tipa
- niz ivalue može da sadrži tri tipa podataka
  1) ceo broj
  2) indeks character promenljive u nizu stringstore
  3) cele brojeve koji su kôdovi pojedinih naredbi
- memoriju čine pet nizova : ivalue,car,cdr, isatom,isnumb. Car i cdr su istog tipa kao i ivalue dok su poslednja dva tipa bit(1) tj. služe kao indikatori
- vrednosti nizova car i cdr su pokazivači na elemente listâ u memoriji
- niz isatom je indikator da li je objekat na koga pokazuju car ili cdr atom
- niz isnumb je indikator da li je objekat na koga pokazuju car ili cdr broj
- podaci se prikazuju na sledeći način :
  - broj : isatom & isnumb (broj je numerički atom)
  - character : isatom & ^isnumb (character je simbolički atom)
  - lista : ^isatom & ^ isnumb
- svi pokazivači su tipa binary fixed(31,0)
- skupljač otpadaka(garbage collector) je tipa mark and collect.

3. Prevodilac i COMPILE funkcija

Prevodilac LispKit Lisp jezika na mašinski jezik SECD mašine se može zadati kao funkcija COMPILE. Definišimo funkciju COMPILE na sledeći način:

$$COMPILE(F)=F^*$$

F je korisnička funkcija napisana na LispKit Lisp jeziku. F^* je prevod funkcije F (napisane na LispKit Lisp jeziku) na mašinski jezik SECD mašine. Funkcija COMPILE može se definisati na LispKit Lisp jeziku na sledeći način:

```
(letrec compile
   (compile lambda(e)
     (comp e (quote nil) (quote (4 21))))
   (comp lambda (e n c)
     (if (atom e) ......
     (if (eq (car e) (quote quote))
         ( ........)
     (if (eq (car e) (quote add))
         ( ........)
     (if (eq (car e) (quote sub))
         ( ........)
        .
        .
        .
     (if (eq (car e) (quote letrec))
         ( ........)
        .
        .
        .
   )
)
```

3.1 Specifičnosti programske implementacije

Prevodilac LispKit Lisp jezika na mašinski jezik SECD mašine napisan je kao korisnička funkcija LispKit Lisp jezika. Ovakav način konstrukcije prevodioca poseduje sledeće dobre osobine:

- program za prevođenje je jako kratak
- omogućeno je jednostavno uvođenje novih funkcija (u LispKit Lisp) i odgovarajućih naredbi u program za prevođenje
- pojednostavljuje se otkrivanje i ispravljanje grešaka u tako dobijenom prevodiocu.

Prevodilac koji se konstruiše je jedna korisnička funkcija (označimo je sa compile1). Prevedimo dobijenu funkciju compile1:

$$compile(compile1)=compile1^*$$

$compile1^*$ je prevod funkcije compile1(proširenog prevodioca) na mašinski jezik SECD mašine. Umesto funkcije compile uzmimo compile1 i ponovimo istu operaciju.

$$compile1^*(compile1)=compile1^{**}$$

U slučaju da je konstruisani (prošireni) prevodilac ispravan njegovi prevodi $compile1^*$ i $compile1^{**}$ moraju biti identični(kao dva prevoda iste funkcije compile1). Da bi sproveli ovakvu proceduru na početku moramo da posedujemo već prevedenu tačnu verziju prevodioca (makar i minimalnu) na mašinski jezik SECD mašine koja će biti osnova za dalja proširenja.

4. Funkcije EXPLODE i IMPLODE

Sastavni deo atoma dužine n (u oznaci S_n ,n>=1) je niska od n slova, cifara ili specijalnih znakova tako da se atom može prikazati u obliku aS_nb gde su a i b niske od m_1 i m_2 znakova ili prazne niske.

4.1 Definicije funkcija EXPLODE i IMPLODE

EXPLODE je unarna funkcija. Argument je valjani S-izraz. Dozvoljena vrednost argumenta je simbolički atom ili nenegativan numerički atom pri čemu se znak +, ako postoji, ignoriše. Vrednost funkcije je lista svih sastavnih delova vrednosti argumenta dužine 1.

```
Primer :  explode((primer))=(p r i m e r)
          explode((13245)) =(1 3 2 4 5)
          explode((+99675)) =(9 9 6 7 5)
```

IMPLODE je unarna funkcija. Argument je valjani S-izraz. Dozvoljena vrednost argumenta je lista nenegativnih numeričkih atoma ili lista atoma čiji je prvi element simbolički atom , a ostali elementi su simbolički ili nenegativni numerički atomi. Vrednost funkcije je atom jednak konkatenaciji redom svih elemenata vrednosti argumenta.

```
Primer :  implode((B e o g r a d))=Beograd
          implode((3 2 5)) =325
          implode((PL / 1))=PL/1
```

4.2 Matematičke osobine funkcija EXPLODE i IMPLODE

Neka je A = {x : x je atom}
L = {y : y je lista čiji su elementi sastavni delovi atoma dužine 1}
L' = {z : z je lista čiji je poslednji element atom nil}

Važi
1) $L \subset L'$
2) explode : A --> L
3) implode : L' --> A
4) na datim skupovima (A i L) explode i implode su bijekcije
5) explode i implode su inverzne funkcije u oblasti definisanosti : $\forall x \in A \ \& \ \forall z \in L'$ važi implode(explode(x))=x i explode(implode(z))=z
6) $z \notin L'$ explode(implode(z))$\neg$=z. Npr. explode(implode((ma ma)))$\neg$=(ma ma)

4.3 Programska implementacija funkcija EXPLODE i IMPLODE

4.3.1 Prevod na mašinski jezik

Definišimo prevod funkcija LispKit Lisp Jezika na mašinski jezik. Neka je n lista imena promenljivih, a e valjani S-izraz. Obeležimo sa e*n prevod S-izraza e za datu listu imena promenljivih n. Razmotrimo izraz:

(EXPLODE e)

on ima tzv. osobinu čistog rezultata [1]. Prvo se izračuna vrednost S-izraza i upisuje na vrh steka S.Zatim se na nju primenjuje expl naredba SECD mašine i dobijenom vrednošću zamenjuje vrednost na vrhu steka S. Na taj način se menja samo sadržaj steka S tj. funkcija ne produkuje bočni efekat. Izrazu (EXPLODE e) odgovara sledeći mašinski kôd:

```
(EXPLODE e)*n = e*n ! (expl)
```

! je oznaka za konkatenaciju. Analogno:

(IMPLODE e)*n = e*n ! (impl) . Izraz (IMPLODE e) takođe ima osobinu čistog rezultata. Naredbe expl i impl se izvršavaju na sledeći način:

```
(a.S) E (expl.C)  D --> (expl(a).s)  E  C  D
(a.S) E (impl.C)  D --> (impl(a).s)  E  C  D
```

4.3.2 Modifikacija prevodioca i SECD mašine

Prevodilac LispKit Lisp jezika na mašinski jezik SECD mašine je proširen sledećim segmentom:

```
      .
      .
      .
      (if (eq (car e) (quote explode))
          ( comp (car (cdr e) ) n
                 (cons (quote 33) c) )
      (if (eq (car e) (quote implode))
          ( comp (car (cdr e) ) n
                 (cons (quote 34) c) )
```

33 i 34 su kôdovi expl i impl naredbi SECD mašine.

4.3.3 Modifikacija SECD mašine

Procedura exec se proširuje blokovima naredbi koji odgovaraju novouvedenim expl i impl naredbama SECD mašine. PL/1 program koji realizuje takvu SECD mašinu je sledećeg oblika:

```
exec : proc returns( binary fixed(31,0) );

t=symbol;
ivalue(t)=store(true);
f=symbol;
ivalue(f)=store(false);
s=cons(args,nil);
e=nil;
d=nil;
c=fn;
rep=nil;

/* dodeljivanje početne vrednosti registrima
   SECD mašine i specijalnim atomima t (true) i
   f (false)   */

do until ivalue(car(c))=21);
   select(ivalue(car(c)));
         when(1) begin;            /*   ld   */
                    .
                    .
                    .
                 end;
           .
           .
           .
         when(21)  ;               /*  stop  */
           .
           .
           .
         when(33)  begin;          /*  expl  */
              if isatom(car(s))
                 then
                 do;
              s=cons(explode(car(s)),cdr(s));
              c=cdr(c);
                 end;
                 else
                 do;
                    /* poruka o grešci */
                    stop;
                 end;
         when(34)  begin;          /*  impl  */
              if ^isatom(car(s))
                 then
                 do;
              s=cons(implode(car(s)),cdr(s));
              c=cdr(c);
                 end;
                 else
                 do;
                    /* poruka o grešci */
                    stop;
                 end;
                  end;
           .
           .
           .
         other begin;
                    /* poruka o grešci */
                    stop;
                 end;
   end;    /*  select  */
end;  /* do until */
return(car(s));
end;  /* exec */
```

Explode i implode su realizovane kao funkcije. Parametar funkcija explode i implode je adresa vrednosti argumenta. Vrednost funkcija je adresa izračunate vrednosti.
U slučaju funkcije explode:
- Ako je vrednost argumenta simbolički atom postupak je sledeći:
  a) izračunava se dužina simboličkog atoma
  b) konstruiše se lista čiji su elementi sastavni delovi dužine 1 tog atoma;
- Ako je vrednost argumenta numerički atom konstruiše se lista čiji su elementi sastavni delovi dužine 1 (cifre).

```
explode : proc(m) returns(binary fixed(31,0));
declare m binary fixed(31,0),
                        /* adresa argumenta */
        pok             binary fixed(31,0),
    /* adresa argumenta koji se dodaje listi */
        povratak        binary fixed(31,0),
                /* adresa rezultata funkcije */
        j               binary fixed(15,0),
        len             binary fixed(15,0),
          /* sadrži dužinu simboličkog atoma */
        sta             character(15)  var,
                    /* sadrži simbolički atom */
        prom            character(15)     ,
    /* baza iz koje se dobijaju delovi atoma */
        prom1           character(15)  var,
/* služi kod dodavanja elemenata tipa char -oni
  su tipa char(15) var kao što je napomenuto */
        broj pic '(14)z9' def  prom pos(1) ,
     /* za dobijanje cifara numeričkog atoma */
        znak(15) char(1) def prom pos(1)   ;
/* za izdvajanje sastavnih delova argumenta.
  Ako su sastavni delovi cifre konverzija će
  biti automatski izvršena pri dodeli brojevnoj
  promenljivoj  */
if issymbol(m)
   then
   do;
      sta=stringstore(ivalue(car(c)));
             /* dobijanje simboličkog atoma */
      len=length(sta);
             /* određivanje njegove dužine */
      prom=sta;
             /* priprema atoma za delenje   */
      rep=nil;
      do j=len to 1 by -1;
         pok=symbol;
             /* u pitanju je simbolički atom */
         prom1=znak(j);
         ivalue(pok)=store(prom1);
             /* dodeljuje mu se vrednost    */
         rep=cons(pok,rep);
             /* dodavanje na početak liste */
      end;
      povratak=rep;
             /* vrednost koja je rezultat  */
      rep=nil;
      return(povratak);
   end;
   else
   do;
      if ivalue(m)<0 then /* poruka o grešci */
      broj=ivalue(m);
    /* raspakivanje argumenta iz oblika binary
fixed(31,0) i dobijanje dekadnih cifara.Broj je
oblika '       xxxxxxxx' gde x-ovi prikazuju
cifre broja(promenljiv broj x-ova u zavisnosti
od veličine argumenta                          */
      rep=nil;
      do j=15 to 1 by -1 while(znak(j)^=' ');
         pok=number;
             /* vrednost argumenta je broj */
         ivalue(pok)=znak(j);
             /* dodeljuje mu se vrednost    */
         rep=cons(pok,rep);
             /* dodaje se na početak liste */
      end;
      povratak=rep;
      rep=nil;
      return(povratak);
   end;
end; /* explode */
```

Funkcije symbol i number vrše zauzimanje nove memorijske lokacije postavljajući istovremeno odgovarajuću bit kombinaciju nizova isatom i isnumb. Ako u tom trenutku nema slobodnih memorijskih lokacija poziva se skupljač otpadaka koji može da uništi do sada konstruisanu listu. Za eliminisanje ove mogućnosti služi promenljiva rep. Ona je globalna za exec (dakle deklarisana u glavnoj proceduri) i markira se pre poziva skupljača otpadaka. Na taj način do sada konstruisana lista ostaje nepromenjena. Pošto nam nije potrebna za druge svrhe van funkcije explode ona ima vrednost nil.
U slučaju funkcije implode postupak se sastoji u konkatenaciji elemenata argument liste.

```
implode : proc(m)  returns(binary fixed(31,0));
declare  m                  binary fixed(31,0),
                          /* adresa argumenta */
         rez     character(30) var initial(''),
/* inicijalizujemo rezultat kao praznu nisku */
         help               binary fixed(31,0),
                /* adresa rezultata funkcije */
         broj               binary fixed(31,0),
               /* numerička vrednost rezul. */
         j                  binary fixed(15,0),
         len binary fixed(15,0) def rez pos(1),
      /* sadrži dužinu rezultata.Automatski se
       postavlja                             */
         rez1               pic '(14)z9'     ,
     /* za dobijanje cifara numeričkog atoma */
        polje(15)    char(1) def rez1 pos(1) ,
   /* za izdvajanje cifara argumenta ako je on
      numerički atom                         */
        nije_nil             bit(1) init('1'b);
  /* indikator da li se došlo do poslednjeg
     atoma ( koji treba da je nil )          */
if issymbol(m) ! isnumb(m)
   then
   if issymbol(m)
   then
   if stringstore(ivalue(m))=
                        stringstore(ivalue(nil)
      then return(nil);  /* implode(nil)=nil */
      else do;
             /* poruka o grešci */
             stop;
          end;
   else do;
          /* poruka o grešci */
          stop;
        end;
   else
   if iscons(car(m))
      then do;
             /* poruka o grešci */
             stop;
          end;
      else
      if issymbol(car(m))
             /* rezultat je simbolički atom */
        then
        do;
           do while(nije_nil & len<=15);
           /* ispitivanje uslova iscons(car(m))
                  i ako nije nastavlja se */
              if issymbol(car(m))
                 then rez=rez!!
                   stringstore(ivalue(car(m)));
                 /* ako je element simbol njega
                   konkateniramo na rezultat */
                 else
                 if izpoz(car(m))
                    then
                    do;
                       rez1=ivalue(car(m));
                            /* raspakujemo ga */
                       do j=1 to 15;
                          if polje(j)^=' '
                             then
                             rez=rez!!polje(j);
                      /* konkateniramo cifre */
                       end;
                    end;
                    else
                    do;
                       /* poruka o grešci */
                       stop;
                    end;
              m=cdr(m);
                    /* sledeći element liste */
              if issymbol(m) ! isnumb(m)
                 then
                 if issymbol(m)
                    then
                    if stringstore(ivalue(m))=
                        stringstore(ivalue(nil)
                       then nije_nil='0'b;
                       else
                       do;
                          /* poruka o grešci */
                         stop;
                       end;
```

```
                else
                doj
                   /* poruka o greški */
                  stop;
                endj
          endj  /* do while */
          if len>15
             then
             doj
                /* poruka o greški */
                stop;
             endj
             else
             doj
                help=symbol;
                        /* simbolički atom */
                ivalue(help)=store(rez);
                              /* vrednost */
                return(help);
             endj
       endj
       else
       doj
            do while(nije_nil & len<15)
                 /* ako je car(m) simbolički
                    atom,tipa cons ili nega-
                    tivan broj daje se izve-
                    štaj o greški i prekida
                    program                */
                rez1=ivalue(car(m));
                           /* raspakivanje */
                do j=1 to 15;
                        /* i konkatenacija */
                   if polje(j) ^= ' '
                     then rez=rez||polje(j);
                end;
                m=cdr(m);
                if issymbol(m) | isnumb(m)
                 /* ostatak liste je atom */
                  then
                  if issymbol(m)
                   then
                   if stringstore(ivalue(m))=
                       stringstore(ivalue(nil))
                      then nije_nil='0'b;
                          /* regularan kraj */
                     else
                     doj
                        /* poruka o greški */
                       stop;
                 /* poslednji atom nije nil */
                      endj
                  else
                  doj
                     /* poruka o greški */
                     stop;
                 /* poslednji atom nije nil */
                   endj
            endj
       /* ako rezultat nije iz intervala
             -214783648,214783647
          poruka o greški i prestanak rada */
       help=number;      /* rezultat je broj */
       rez1=rez;
  /* rezultat dodeljujemo brojevnoj promen. */
       broj=rez;
          /* pakujemo sa u bin fixed(31,0) */
       ivalue(help)=broj;
       return(help);
izpoz : proc(n) returns( bit(1) );
dcl n binary fixed(31,0);
    return( ivalue(n))=abs(ivalue(n)) );
endj /*  izpoz  */
endj /* implode */
```

Ovakav način implementacije IMPLODE bio je uslovljen mogućim argumentima funkcije. Ako je argument IMPLODE rezultat funkcije EXPLODE koja je za svoj argument imala numerički atom logično je da i rezultat bude takav. Međutim bilo koja operacija nad rezultatom EXPLODE ne menja tip podatka.Tako je IMPLODE (cdr (EXPLODE (a12345)))='12345' kao simbolički a ne numerički atom.

5. Primena

Često se u upotrebi susreću funkcije ATOMCAR i ATOMCDR.One se mogu definisati,preko funkcija EXPLODE i IMPLODE, na sledeći način:

```
-   (ATOMCAR LAMBDA(X)
          (CAR (EXPLODE X)) )
-   (ATOMCDR LAMBDA(X)
          (IMPLODE (CDR (EXPLODE X))) )
```

Navedimo četiri primera koji ilustruju primenu funkcija EXPLODE i IMPLODE.

Primer 1.

Definisati unarnu funkciju PAL. Vrednost argumenta je simbolički atom. Vrednost funkcije je t ako je taj atom palindrom, u suprotnom f (palindrom je reč koja se isto piše i sleva udesno i zdesna ulevo).

Na primer:

```
PAL(anavolimilovana)=t
PAL(palindrom)=f
```

Sledeći program je rešenje zadatka:

```
 (LETREC PAL
   (PAL LAMBDA(X)
      (IF (ATOM X)
          (EQUAL (REVERSE (EXPLODE X) )
                 (EXPLODE X) )
          (QUOTE F)
      ))
   (REVERSE LAMBDA(X)
      (REV X (QUOTE NIL) ) )
   (REV LAMBDA(X Y)
      (IF (EQ X (QUOTE NIL) )
          Y
          (REV (CDR X) (CONS (CAR X) Y) )
      ))
   (EQUAL LAMBDA(X Y)
       ( IF (ATOM X)
            (EQ X Y)
            (IF (ATOM Y)
                (EQ X Y)
                (IF (EQUAL (CAR X) (CAR Y) )
                    (EQUAL (CDR X) (CDR Y) )
                    (QUOTE F)
                )
             )
        ) ) )
```

Primer 2.

Definisati unarnu funkciju KONT. Vrednost argumenta je numerički atom. Vrednost funkcije je t ako vrednost argumenta sadrži najmanje dve iste cifre, a f u suprotnom.

Na primer:

```
KONT(1232567)=t
KONT(-723269)=t
KONT(-723469)=f
KONT(8723469)=f
```

Zadatak ćemo rešiti na dva načina:

a) Primenom EXPLODE funkcije:

```
 (LETREC KONT
    (KONT LAMBDA ( X )
      (IF (LEQ X (QUOTE 0) )
          (ISPITAJ (EXPLODE (SUB (QUOTE 0) X)))
          (ISPITAJ (EXPLODE X) )
       ))
    (ISPITAJ LAMBDA ( NEISPITANO )
       (IF (EQ NEISPITANO NIL)
           (QUOTE F)
           (IF (MEMBER (CAR NEISPITANO)
                       (CDR NEISPITANO) )
               (QUOTE T)
               (ISPITAJ (CDR NEISPITANO) )
           )))
```

```
(MEMBER LAMBDA (E N)
      (IF (EQ N (QUOTE NIL) )
          (QUOTE F)
          (IF (EQ E (CAR N) )
              (QUOTE T)
              (MEMBER E (CDR N) )
          ))))
```

b) Primenom aritmetičkih funkcija:

```
(LETREC KONT
   (KONT LAMBDA ( X )
      (IF (LEQ X (QUOTE 0) )
          (KONT1 (SUB (QUOTE 0) X) )
          (KONT1 X )
      ))
   (KONT1 LAMBDA ( X )
      (IF (LEQ X (QUOTE 9) )
          (QUOTE F)
          (IF (ISPITAJ (REM X (QUOTE 10) )
                       (DIV X (QUOTE 10) ) )
              (QUOTE T)
              (KONT1 (DIV X (QUOTE 10) ) )
          )))
   (ISPITAJ LAMBDA (CIFRA BROJ )
      (IF (LEQ BROJ (QUOTE 9) )
          (EQ CIFRA BROJ)
          (IF (EQ CIFRA
                  (REM BROJ (QUOTE 10) ) )
              (QUOTE T)
              (ISPITAJ CIFRA
                       (DIV BROJ (QUOTE 10) ) )
          ))))
```

Primer 3.

Definisati unarnu funkciju MARK. Vrednost argumenta je lista atoma koja može da sadrži iste specijalne atome - markere. Vrednost funkcije je prazna lista ili lista atoma dobijenih konkatenacijom atoma koji se nalaze između:

- početka liste i prvog markera,
- dva susedna markera,
- poslednjeg markera i kraja liste,
- početka i kraja liste (ako lista ne sadrži marker).

Markeri se izostavljaju. Za marker uzeti *.

NA primer:

```
MARK((a 2 3 * 1 2 * * 12 * a s d* a * ** i))=
            (a23 12 12 asd*a **i)
MARK((PRI   MER * 3 * FUNK   CIJA * MARK * ))=
             ( PRIMER 3 FUNKCIJA MARK )
```

Sledeći program je rešenje zadatka:

```
(LETREC MARK
   (MARK   LAMBDA(X)
     (REVERSE (IZBACI NIL X NIL) ))
   (IZBACI LAMBDA (ARG X REZ)
     (IF (EQ X NIL)
         (IF (EQ ARG NIL)
             REZ
             (CONS (IMPLODE (REVERSE ARG)) REZ)
         )
         (IF (EQ (CAR X) (QUOTE *))
             (IF (EQ ARG NIL)
                 (IZBACI NIL (CDR X) REZ)
                 (IZBACI NIL (CDR X)
                         (CONS (IMPLODE
                                 (REVERSE ARG))
                               REZ)))
             (IZBACI (CONS (CAR X) ARG)
                     (CDR X) REZ )
         )))
   (REVERSE LAMBDA(X)
      (REV X (QUOTE NIL) ) )
   (REV LAMBDA(X Y)
      (IF (EQ X (QUOTE NIL) )
          Y
          (REV (CDR X) (CONS (CAR X) Y) ) )
      )))
```

Primer 4.

Definisati unarnu funkciju FIND. Vrednost argumenta je lista atoma. Vrednost funkcije je lista neizmenjenih atoma izuzev onih čiji su prvi sastavni delovi dužine 1 simboli < ili > . Ovi atomi se dele na dva atoma: < (ili >) i ostatak atoma (funkcija FIND ima primenu kod složenih MATCH funkcija).

Na primer:

```
FIND((adssf a <aasd >fhsn >> sdf> rf< >)<<<))=
(adssf a < aasd > fhsn > > sdf> rf< > <<<))
FIND(( beograd >sava <dunav avala )=
     ( beograd > sava < dunav avala )
```

Sledeći program je rešenje zadatka:

```
(LETREC FIND
  (FIND LAMBDA(X)
    (FLATTEN (REVERSE (IZBACI X (QUOTE NIL))))
  )
  (IZBACI LAMBDA(ULAZ IZLAZ)
     (IF (EQ ULAZ (QUOTE NIL) )
         IZLAZ
         (IF (NUMBERP (CAR ULAZ) )
             (IZBACI (CDR ULAZ)
                     (CONS (CAR ULAZ) IZLAZ) )
             (LET (IF (AND (OR (EQ (CAR PRVI)
                                   (QUOTE <))
                               (EQ (CAR PRVI)
                                   (QUOTE >)))
                           (NOT (EQ (CDR PRVI)
                                    (QUOTE NIL))))
                      (IZBACI
                        (CDR ULAZ)
                        (CONS
                          (CONS (CAR PRVI)
                            (CONS (IMPLODE
                                    (CDR PRVI))
                                  (QUOTE NIL)))
                          IZLAZ))
                      (IZBACI (CDR ULAZ)
                              (CONS (CAR ULAZ)
                                    IZLAZ)))
                  (PRVI EXPLODE (CAR ULAZ) )
             ))))
  (FLATTEN LAMBDA(X)
     (IF (EQ X (QUOTE NIL))
         (QUOTE NIL)
         (IF (ATOM (CAR X))
             (CONS (CAR X) (FLATTEN (CDR X)))
             (APPEND (FLATTEN (CAR X))
                     (FLATTEN (CDR X))
             ))))
  (APPEND LAMBDA(X Y)
     (IF (EQ X (QUOTE NIL))
         Y
         (CONS (CAR X) (APPEND (CDR X) Y))))
  (REVERSE LAMBDA(X)
    (REV X (QUOTE NIL) ) )
  (REV LAMBDA(X Y)
     (IF (EQ X (QUOTE NIL) )
         Y
         (REV (CDR X) (CONS (CAR X) Y) ) ))
  (NUMBERP LAMBDA(X)
    (EQ X (ADD X (QUOTE 0) ) ) ))
```

6. Zaključak

Pojam atom označava nedeljiv objekat. Međutim, uvođenjem funkcija EXPLODE i IMPLODE, izvršen je prodor u strukturu atoma. Atom je postao deljiv do na dužinu 1. Atomi dužine 1 su nedeljivi. Ovo je od posebnog značaja za simboličke atome jer su se numerički atomi pomoću celobrojnog deljenja mogli i ranije rastavljati na sastavne delove (cifre). Navedene funkcije imaju veliku primenu u obradi teksta. One se ne mogu realizovati kao korisničke funkcije već isključivo kao ugrađene funkcije na nivou SECD mašine.

Literatura

[1] Peter Henderson
Functional Programming!
application and implementation
Prentice-Hall International, inc.,
London, 1980.

[2] Patrick Henry Winston &
Berthold Klaus Paul Horn
Lisp
Addison-Wesley Publishing Company,
Reading, Massachusetts, 1981.

[3] Patrick Henry Winston
Artificial intelligence
Addison-Wesley Publishing Company,
Reading, Massachusetts, 1977.

[4] J. Darlington, P. Henderson, D. A. Turner
Functional programming and its application
Cambridge University Press,
Cambridge, 1982.

[5] V. Stojković i drugi
Lisp-Kit Lisp jezik, verzija ARL
Bilten za nauku i informatiku SAPV, 1984.

Zahvaljujemo se koleginici Mariji Kulaš iz Instituta za matematiku u Novom Sadu na sugestijama.

INFORMATICA 2/85

ANALYSIS OF CODE GENERATION FOR A COMMUTATIVE ONE-REGISTER MACHINE

DUŠAN PETKOVIĆ

SIEMENS AG, MÜNCHEN, F. R. GERMANY

UDK: 681.3.325.6

ABSTRACT. This paper presents a development of the analysis of code generation for the set of expressions with common subexpressions for the commutative one-register machine. A heuristic based on this development is shown to produce code size better than 3/2 that of an optimal coding for any expression. This result underestimates the results known in the literature.

ANALIZA GENERISANJA KODA ZA KOMUTATIVNU MAŠINU SA JEDNIM REGISTROM - U radu je data analiza generisanja koda za skup izraza sa zajedničkim podizrazima za komutativnu mašinu sa jednim registrom. Heuristika zasnovana na ovom razvoju daje kod čiji količnik u odnosu na optimalan kod je bolji od 3/2. Tako dobijeni rezultat predstavlja poboljšanje u odnosu na rezultate dobijene u drugim radovima.

1. INTRODUCTION

We consider the problem of generating optimal code for a set of expressions. If the set of expressions has no common subexpressions then a number of efficient optimal code generation algorithms are known (Sethi, Ullman (1970) and Aho, Johnson (1976)). However, generating optimal code for expressions with common subexpressions for a one-register machine has been shown to be NP- complete by Bruno and Sethi (1970). Aho, Johnson and Ullman (1976) have shown that the problem remains NP- complete even for simple expressions. They also developed a heuristic which has a worst case ratio of 3/2 for one-register (noncommutative)machine. In the same paper they discussed code generation for commutative machines and showed that 3/2 is a worst case ratio for the generalized top-down greedy algorithm for a commutative one-register machine.

At the end of that paper Aho, Johnson and Ullman proposed some open questions, and one of these open questions is: how closely can a polynomial time heuristic approximate optimal code generation problem on a commutative one-register machine?

This paper presents a uniform development of a linear algorithms for the commutative one-register machine along with an analysis of bounds. A heuristic based on this development is shown to produce code size better than 3/2 that of an optimal coding, using only optimal number of computational and store instructions.

2. BACKGROUNDS AND DEFINITIONS

In translating a source program into machine language most compilers first translate the source program into an intermediate form,which is then subsequently transformed into the final object program. The intermediate code represents a flow graph where each node represents straight line block (basic block). Within a basic block the flow of control is sequential. Each basic block in a flow graph can be represented by a DAG (directed acyclic graph - Aho, Ullman (1971)).

Figure 1 shows a basic block and its corresponding DAG.

$n_1 \leftarrow d + e$
$n_2 \leftarrow b + c$
$n_3 \leftarrow n_2 * n_1$
$n_4 \leftarrow n_3 - n_1$
$n_5 \leftarrow a - n_2$
$n_6 \leftarrow n_5 + n_3$

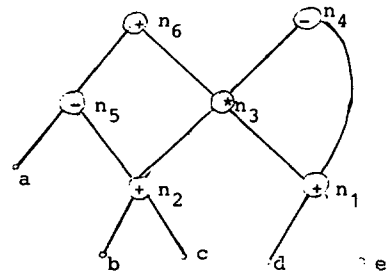


Fig. 1

We call node n_3 a right child of node n_6, and a left child of node n_4. Accordingly, we call node n_3 a right parent of node n_2 and node n_5 a left parent of node n_2.

A node with no children is called a leaf. A node with no parents is called a root. Interior nodes are nodes that are not leaves. A tree is a DAG, where each node (except roots) has always only one parent (left or right).

A machine program is a sequence of instructions. The length of a program is the number of instructions it contains.

One-register (noncommutative) machine is limited to three basic types of instructions:

r←m (load instruction),
r←r op m (op- instruction)
and m←r (store instruction),

where r is register, and m is any memory location (a noncommutative machine has the left operand always in register and the right operand is always in memory).

A generalization of this machine is a commutative one-register machine. Commutative one register machine is one where, for each instruction of the form

$$r \leftarrow r \text{ op } m,$$

there is also an instruction of the form

$$r \leftarrow m \text{ op } r.$$

That means, the order of the children of any node is permutable, as far as code generation is concerned, although the operator itself may not be commutative.

3. HEURISTIC TECHNIQUE

The following assumptions are made:

1. Code generation is limited to that for a commutative one-register machine.
2. All generators produce no useless instructions, i. e. instructions which can be deleted without changing the value of the program.
3. All generators never store the same value more than once.
4. Input to the code generators is a DAG,with possible multiple roots, each of which has to be stored.
5. Commutativity and associativity of expressions are not conserned in this paper.

Assumptions 2. and 3. restrict the class of all programs which evaluate any DAG. This restriction serves only to eliminate very inefficient programs. From any program P we can easily construct an equivalent program which fulfilles the assumptions 2. and 3. (in the time proportional to the length of P).

Theorem 1. The function f: A→B, where A is a set of all computational instructions required in an optimal coding of a particular DAG, and B is a set of all internal nodes of the same DAG, is bijective.

Proof. Directly from the assumption 2 above.

Before we state the theorem about the number of store instructions, required in an optimal coding, we need further definitions and algorithms.

Definition 1. A left chain is a tree with following property: each interior node of a tree has always a leaf as a right child.

Definition 2. A worm is a tree with following property: each interior node of a tree has at least one leaf as a child.

Each left chain is a worm, but each worm must not be left chain.

Algorithm 1A. Given a DAG D, a forest of trees T may be constructed by pruning all interior nodes with multiple parents from each of their parent node in the DAG. Each resulting tree t_i is labeled, identifying the leaves that replace the pruned sub-DAGs. The forest of trees from a given DAG is unique.

Algorithm 1A gives us a subset of all nodes which has to be stored in an optimal coding (it is obvious that all internal nodes with multiple parents has to be stored).

Besides them, we need to store some nodes of a pruned trees. If the pruned tree t_i is a worm we need not to store any node of that tree. If the pruned tree is not a worm we need to store at least one internal node of that tree. Further, the commutative machine allows us to think of all operators to be commutative. Sethi and Ullman (1970) showed (in Algorithm 2 of that paper) which internal nodes has to be stored in generating optimal code for the tree with commutative operators. Using the result of that paper we can state the following

Algorithm 1B.

Step 1. Take a tree t_i from the algorithm 1A (there must be at least one such tree, if the particular basic block has common subexpressions). Go to step 2.

Step 2. If t_i is a worm, go to step 3. If t_i is not a worm, apply algorithm 2 from Sethi and Ullman (1970) to the tree t_i to get all internal nodes which have to be stored. Mark those nodes. Prune each marked node from his parent node. Each resulting sub-tree t_{ij} is a unique left chain. Go to step 3.

Step 3. Take a next tree t_k from the algorithm 1A and apply step 2. If there is no such tree end.

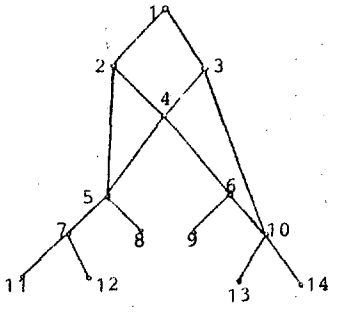
Algorithm 1B gives us a second subset of nodes which has to be stored in an optimal coding.

From the discussion above we can state following theorem:

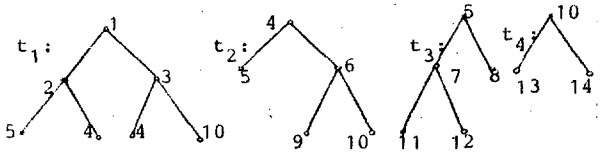
Theorem 2. The number of store instructions required in an optimal coding of a particular DAG for the commutative one-register machine is the sum of:

a) all roots;
b) all internal nodes with multiple parents;
c) all marked nodes by the algorithm 1B.

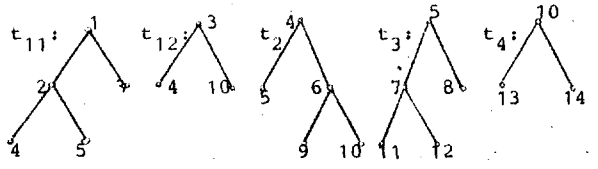
Figure 2 shows pruning DAG D, first using algorithm 1A and afterwards using algorithm 1B (2a shows a DAG D, 2b shows the pruned sub-DAGs t_i after applying algorithm 2A to DAG D, and 2c shows the pruned sub-trees t_{ij} after applying algorithm 2B to sub-DAG t_i).



2a)



2b)



2c)

From now on, we use the results of Carter (1982) to get the wishing heuristic. That means the rest of this paper is equivalent to the paper of Carter, however we concern the commutative one-register machine instead of noncommutative one.

Definition 3. A Data Flow Dependency graph (DFDEP) is constructed from the forest W of worms derived from a DAG D after applying the algorithms 1A and 1B. The vertices of the DFDEP graph correspond to the worms of W. An arc from vertex w_i to vertex w_j specifies that the worm w_i uses the result of the worm w_j as an operand.

Theorem 3. The DFDEP graph from a forest W of worms derived from a DAG D is itself a DAG.

Proof. Since D is DAG, and the forest W only groups directly connected nodes of D into worms without introducing new arcs, the DFDEP graph must be acyclic.

Definition 4. A Data Flow Ready set (DFREADY) is the set of vertices from the DFDEP graph with no outward arcs.

The elements of the DFREADY set correspond to those worms whose operands have already been coded or were leaves in the original DAG D and are therefore ready to be coded. By the acyclic nature of the DFDEP graph, the DFREADY set will always have at least one member, as long as there is at least one worm to be coded.

The process of coding a DAG may be viewed as a multi-step algorithms. First, a forest of worms is constructed from the DAG. Second, a DFDEP graph is constructed, and a DFREADY set from it.

The final step is an iterative coding. A member of the DFREADY set is removed along with the corresponding vertex in the DFDEP graph. All inward arcs are also removed from that vertex. Any vertices now without outward arcs are added to the DFREADY set, and the worm corresponding to the removed DFREADY element is coded, ending with the store operation. This iteration continues until the DFREADY set is empty meaning that the DAG is coded.

The DFDEP graph and DFREADY set should be viewed as dynamic data structures, which are modified during the course of code generation. A code generation could be characterized by the algorithm it uses in constructing the forest of worms, and the transformations it performs on the DFDEP graph. (The DFREADY set is always derivable from the current DFDEP graph so it is not required as part of characterization.)

Let C be a class of code generators which produce code when forest W is constructed by algorithm 1A and algorithm 1B, the DFDEP graph and the DFREADY set. The difference between members of the class is the method which is used to select from the DFREADY set. The order of selection determines the number of load instructions for internal nodes required in the coding. (The number of leaf loads can be easily shown to be the number of worms whose left operand is a leaf in the original DAG D.)

Lemma 1. Given a DAG D, it is possible to construct the forest W (as specified in the algorithms 1A and 1B), and the DFDEP graph in linear time.

Proof. For the construction of trees t_i (as output of algorithm 1A), we need only two transversals of DAG D, first marking all vertices with multiple parents and building trees t_i second. Building trees is similar to a left-to-right, depth-first DAG copy, where marked vertices indicate where to create new trees. Once created, the marked vertex is flagged to prevent creation of multiple copies of shared sub- DAGs.

The construction of second subset of nodes with algorithm 1B and the construction of the DFDEP graph is similar.

Theorem 4. All of the code generators in C produce code which is better than 3/2 the size of an optimal coding of any DAG D, and at least one generator runs in linear time.

Proof. This proof shows the bound, and the existence of the linear time code generator.

Let S_k be the number of internal nodes in each worm w_k, where $S_k \geq 1$, and n is the number of worms. The result of each worm must be stored, so the number of computational instructions and store instructions is:

$$\sum_{k=1}^{n} (S_k+1).$$

To maximize the ratio of generated code to optimal code, assume that optimal coding requires only one load instruction for each worm:

$$\frac{SIZE(C_i(D))}{SIZE(Optimal(D))} = \frac{\sum_{k=1}^{n} (S_k+2)}{\sum_{k=1}^{n} (S_k+1) + 1} =$$

$$= \frac{\sum_{k=1}^{n} (S_k+1)+1+n-1}{\sum_{k=1}^{n} (S_k+1) + 1} = 1 + \frac{n-1}{\sum_{k=1}^{n} (S_k+1)+1} \leq$$

$$\leq 1 + \frac{n-1}{2n+1} < \frac{3}{2}.$$

Lemma 1 showsthat DFDEP graph construction can be done in linear time. DFREADY set construction and selection can be simulated by using worms in a post-order encounter during a depth-first traversal of the DFDEP graph. Therefore, there is at least one code generator in C which runs in linear time. The result is code, which uses the optimal number of computational and store instructions, thereby insuring object code better than 3/2 the size of an optimal coding.

4. CONCLUSIONS

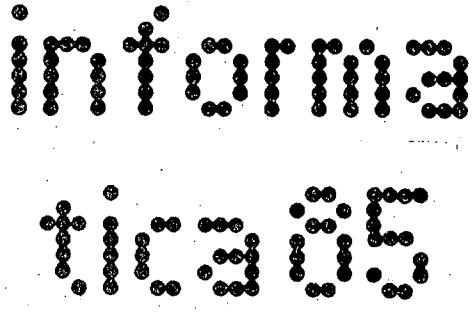
This paper shows a first step to the uniform development of the analysis of code generation for a commutative one-register machine. First, we showed the theorem which gives us the number of computational instructions, required in an optimal coding of a particular DAG with commutative one-register machine. After that, we showed the theorem which gives us the number of store instructions required in an optimal coding of a particular DAG with commutative one-register machine. This theorem is based on two algorithms, algorithms 1A and 1B. These two algorithms are dependent from each other; that means, the output of algorithm 1A is the input of algorithm 1B. Then, using the results of Carter we showed the final theorem, which gives us the wishing result.

This analysis of code generation for a commutative one-register machine uses only optimal number of computational and store instructions. In this paper we did not concern at all optimizing of load instructions. If the load instructions would be optimized, we would get

a heuristic, which gives us better results than this one. To get a better heuristic will be the aim of the future work.

5. REFERENCES

1) Aho, A.V. and Ullman, J.D. - Optimization of Straight Line Programs, SIAM Journal of Computing, Vol. 1, No. 1,pp. 1-19, 1972 .
2) Aho, A.V. and Ullman, J.D. - The Theory of Parsing, Translation and Compiling, Vol. II: Compiling, Prentice Hall, 1973.
3) Aho, A.V., Hopcroft, J.E. and Ullman, J.D.- The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
4) Aho, A.V. and Johnson, S.C. - Optimal Code Generation for Expression Trees, Journal of ACM, Vol. 23, No. 3, pp. 488-501, 1976.
5) Aho, A.V., Johnson, S.C. and Ullman, J.D. - Code Generation for Expressions with Common Subexpressions, Journal of ACM, Vol. 24, No. 1, pp. 146-160, 1977.
6) Aho, A.V. and Ullman, J.D. - Principles of Compiler Design, Addison-Wesley, 1977.
7) Aho, A.V. and Sethi, R. - How Hard is Compiler Code Generation, in: Salomaa, A. and Steinby, M. - Automata, Languages, Programming, Lecture Notes in Compiler Sciences, No. 52, pp. 1-15, 1977.
8) Anderson, J.P. - A Note on Some Compiling Algorithms, Comm. of ACM, Vol. 7, No. 3, pp. 149-150, 1964.
9) Bruno, J. and Sethi, R. - Code Generation for a One-register Machine, Journal of ACM, Vol. 23, No. 3, pp. 502-510, 1976.
10) Carter, L.R. - Further Analysis of Code Generation for a Single Register Machine, Acta Informatica, Vol. 18, pp. 135-147,1982.
11) Nakata, I. - On Compiling Algorithms for Arithmetic Expressions, Comm. of ACM, Vol. 10, No. 8, pp. 492-494, 1967.
12) Redziejowski, R.R. - On Arithmetic Expressions and Trees, Comm. of ACM, Vol. 2, pp. 81-84, 1969.
13) Sethi, R. and Ullman, J.D. - The Generation of Optimal Code for Arithmetic Expressions, Journal of ACM, Vol. 17, No. 4, pp. 715-28, 1970.
14) Stockhausen, P.F. - Adapting Optimal Code Generation for Arithmetic Expressions to the Instruction Sets Available on Present-Day Computers, Comm. of ACM, Vol. 16, No. 6, pp. 353-354, 1973.
15) Waite, W.M. - Optimization, in Bauer, F.L. and Eickel, J. (ed.) - Compiler Construction: An Advanced Course, Springer Verlag, pp. 549-602, 1974.



INFORMATICA '85
61116 Ljubljana, p.p. 2
JUGOSLAVIJA

PAPER/SHORT PAPER/ TECHNICAL REPORT REGISTRATION

This application should be typewritten

1. Title of the Paper: .....
2. Extended summary (approximately 1000 words) should be enclosed.
3. Program Category of the Paper (circle appropriate choice)
   1. Survey of Technology and/or Usage
   2. System Architecture
   3. Process Control
   4. Systems Development Tools
   5. Small Business Systems
   6. Usage in Education
   7. Personal Computers
   8. CAD/CAM Systems
   9. Artificial Intelligence and Robots
   10. Computer Networks
   11. Fifth Computer Generation
4. Classification of the Paper (circle appropriate choice)
   1. Paper
   2. Short paper
   3. Technical report

5. Author(s): .....

   Organization: .....

   Street: .....

   Postal Code: ..... City: .....

   Country: .....

   Date: ..... Signature: .....

This application form together with two copies of extended summary must reach the following address before April 1, 1985: Informatica '85, 61116 Ljubljana, p.p. 2, Yugoslavia

PRIJAVA REFERATA / KRATKEGA REFERATA / STROKOVNEGA POROČILA

Prijavo izpolnite s pisalnim strojem

1. Naslov referata .....
2. Razširjeni povzetek (približno 1000 besed) priložite prijavi.
3. Programsko področje referata (obkrožite ustrezno številko)
   1. Pregled tehnologije in uporabe
   2. Arhitektura in zgradba računalniških sistemov
   3. Upravljanje procesov
   4. Sistemski razvojni pripomočki
   5. Mali poslovni sistemi
   6. Uporaba pri izobraževanju
   7. Osebni računalniki
   8. CAD/CAM mikrosistemi
   9. Umetna inteligenca in roboti
   10. Računalniške mreže
   11. Peta računalniška generacija
4. Razvrstitev referata (obkrožite)
   1. referat (pomembnejše delo)
   2. kratek referat
   3. poročilo

5. Avtorji: .....

   Delovna organizacija: .....

   Ulica: .....

   Poštna številka: ..... Kraj: .....

   Država: .....

   Datum: ..... Podpis: .....

Prijavnica, skupaj z dvema kopijama razširjenega povzetka, mora prispeti najkasneje do 1. aprila 1985 na naslov: Informatica '85, 61116 Ljubljana, p.p. 2

IBMOVSKI OSEBNI RAČUNALNIK PETRA III

ANTON P. ŽELEZNIKAR

UDK: 681.3.06 Petra III

Ta del članka opisuje začetno preizkušanje računalnika Petra, pomen začetnih nastavitev konfiguracijskih stikal, začetne nastavitve skakačev, nastavitev frekvence VCO za krmilnik upogljivih diskov in osnovni V_I sistem (ROM BIOS) za ibmovski mikroračunalnik (z nekaterimi podatki o operacijskem sistemu MS-DOS).

Petra - An IBM-like Personal Computer III

This part of the article deals with initial testing of Petra computer, with meaning of configuring switches, with initial jumper settings, tuning of VCO frequency for floppy disk controller and with basic I_O system (ROM BIOS) for an IBM-like computer (describing some basic concepts of MS-DOS).

11. Začetno preizkušanje

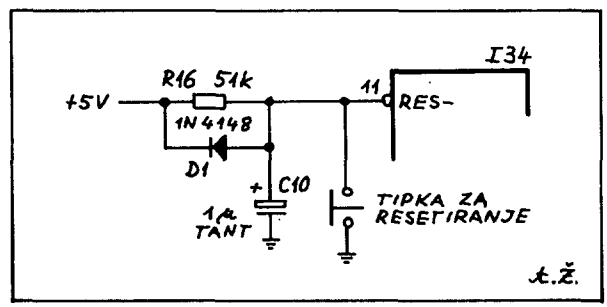
Ko smo ploščo (ali plošče, če jih je več) z integriranimi vezji do konca povezali (izdelali, ožičili, pospajkali), jo (ali jih) začnemo preizkušati v temle zaporedju:

(1) Izmerimo vse napajalne napetosti z voltmetrom, merimo pa tudi šum (ki je lahko posledica slabega filtriranja) na napajalnih sponkah posameznih integriranih vezij z osciloskopom.

(2) Preverimo delovanje vseh oscilatorjev, in sicer na sponkah 8 (sysclk0), 12 (sysclk1) in 2 (sysclk2) vezja I 34 (8284A, list 1), na sponki 8 vezja I 10 (M1116-8M, list 5) in na sponkah 10 (1,2288 Mhz) in 11 (uartclk) vezja I 69 (74 LS 393, list 7).

(3) Preverimo delovanje resetiranja in vgradimo dodatno tipko, kot prikazuje slika 9. Na sistemskem naslovnem vodilu preverimo vse naslovne vode, in sicer na sponkah Q vezij I 44, I 38 in I 50 (list 1). Pri resetiranju se posamezni registri nastavijo takole:

register	vsebina
IP (ukazni kazalec)	0000h
CS (kodni segment)	0ffffh
DS (podatkovni segment)	0000h
SS (skladovni segment)	0000h
ES (posebni segment)	0000h
statusni register	resetiran



Slika 9. Dograditev tipke za resetiranje, tako da je omogočeno začetno preozkušanje sistema in ročno resetiranje sistema, kadar sistem "obvisi".

Na naslovnem vodilu se tako takoj po resetiranju skladno s stanjem IP in CS v razpredelnici pojavi naslov

OffffOh
ali
1111_1111_1111_1111_0000

ki ga preizkusimo z osciloskopom na posameznih naslovnih vodih (dvajsetih), tako da ponavljamo pritiskanje na resetirno tipko.

(4) Po preizkusih točk (1), (2) in (3) začnemo inicializirati posamezne krmilnike, ko vpisujemo različne programe v ROM. Ta ROM vstavimo v I 85 in na pomnilni lokaciji OffffOh (oziroma CS:IP = ffff:0000) imamo (intersegmentni) skok na začetek inicializacijskega programa (o tem kasneje).

12. Začetne nastavitve konfiguracijskih stikal

Nastavitev konfiguracijskih stikal pod oznako ST na listu 7 moramo predpisati (definirati). Tako bo s položajem posameznih stikal določena uporaba enega, dveh, treh ali štirih diskovnih enot dimenzij 5,25 ali 8 col, V_I hitrosti (izražene v baudih), uporaba vgrajenih konzolnih kanalov ali posebnih vtičnih enot (ibmovska tipkovnica in ibmovski monokromatični ali barvni vmesnik). Pri izdelavi tkim. ROM BIOSa (Basic I_O System) ločimo dve osnovni možnosti nastavitev konfiguracijskih stikal, in sicer:

- Imamo sistem z vgrajenim konzolnim kanalom (kanal 0 s priključnico tipa RS-232 in z oznako P 13 na listu 7 prek vezja tipa USART 8251A, tj. I 70 na listu 7).

- Imamo sistem z ibmovskim tipkovničnim vmesnikom in z video vmesnikom (to sta posebni tiskani plošči, ki ju vtaknemo v proste priključnice P 1, ... P 9 na listu 2).

Tako dobimo prvi primer nastavitve konfiguracijskih stikal v tabeli 1.

Tabela 1

številka stikala	nastavitev stikala	funkcija stikala
1 2	0 1	nastavitev števila diskovnih enot: 2 enoti
3	0	enota 5,25 cole (ne 8)
4	0	dvojna zapisna gostota
5	0	vgrajeni kanal RS-232
6 7 8	0 0 0	nastavitev hitrosti terminalskega kanala, in sicer 9600 baud

Drugi primer nastavitve konfiguracijskih stikal je prikazan v tabeli 2.

Tabela 2

številka stikala	nastavitev stikala	funkcija stikala
1 2	1 1	nastavitev števila diskovnih enot: 1 enota
3	1	diskovna enota 8 col
4	1	enojna zapisna gostota
5	1	tipkovnični_video vmes
6 7	1 0	barvni vmesnik tipa 80 X 20
8	0	neuporabljeno

Za konfiguracijska stikala ST na listu 7 naj tako velja tole:

ST1, ST2 — število uporabljenih diskovnih enot
ST3 — diskovna enota 5,25 cole ali 8 col
ST4 — enojna ali dvojna zapisna gostota
ST5 — terminal tipa RS-232 ali tipkovnični_video vmesnik
ST6, ST7, ST8 — pomen nastavitve teh stikal je odvisen od nastavitve ST5; če je ST5 = 0 (izključeno), določajo ST6, ST7 in ST8 kanalski takt od 110 do 9600 baud terminala tipa RS-232; če je ST5 = 1 (vključeno), določata ST6 in ST7 video način, ST8 pa ni uporabljeno

Podrobne stikalne funkcije pa so tele:

Stikali ST1 in ST2 določata število diskovnih enot:

ST1	ST2	število diskovnih enot
1	1	1
0	1	2
1	0	3
0	0	4

Stikalo ST3 določa dimenzijo diska:

ST3	dimenzija diska
0	5,25 cole
1	8 col

Stikalo ST4 določa zapisno gostoto:

ST4	zapisna gostota
0	dvojna gostota (disk 5,25 cole)
1	enojna gostota (disk 8 col)

Stikalo ST5 določa terminalski tip:

ST5	terminalski tip
0	konzola RS-232 (navadni terminal)
1	tipkovnični in video vmesnik

Stikali ST6 in ST7 določata pri ST5 = 1 video način (vtične plošče za tipkovnico, monokromatični in_ali barvni monitor):

ST6	ST7	video način
0	0	monokromatični vmesnik 80 X 25
0	1	barvni vmesnik 40 X 25
1	0	barvni vmesnik 80 X 25
1	1	neveljavno

Tabela 3

skakač	list	pomen			
		nastavitev EPROMov I82,83,84,85:			
		2k	4k	8k	16k
SK1	3	1	0	0	0
SK2	3	0	1	1	1
SK3	3	1	1	0	0
SK4	3	0	0	1	1
SK5	3	1	0	0	0
SK6	3	0	1	1	1
SK7	3	omogočitev uporabe ROMa; nastavitev čakalnega stanja			
SK15A	5	taktna frekvenca, disk 5,25 cole			
SK8	5	taktna frekvenca, disk 8 col			
SK9	5	časovna konstanta, disk 8 col			
SK10	5	časovna konstanta,disk 5,25 cole			
SK11	5	časovna konstanta, disk 8 col			
SK12	5	časovna konstanta,disk 5,25 cole			
SK13	5	taktna frekvenca, disk 5,25 cole			
SK14	5	taktna frekvenca, disk 8 col			
SK15	5	regulacijska nap, disk 8 col			
SK16	5	regulacijska nap, disk 5,25 cole			
SK16A	5	taktna frekvenca, disk 5,25 cole			
SK17	5	t ktna frekvenca, disk 8 col			
SK18-1	6	uporaba motorjev št. 2 in 3			
SK18-2	6	eno_dvostranska diskovna enota			
SK18-3	6	uporaba motorja št. 1			
SK18-4	6	uporaba motorja št. 0			
SK18-5	6	uporaba glave št. 0			
SK18-6	6	konstanten signal READY-			
SK18-7	6	uporaba glave št. 1			
SK18-8	6	uporaba glave št. 2			
SK18-9	6	uporaba glave št. 3			
SK20	5	prednastavitev VCO frekvence, disk 8 col			
SK21	5	prednastavitev VCO frekvence, disk 5,25 cole			
R14	5	natančna nastavitev VCO frekven.			

Stikala ST6, ST7 in ST8 določajo pri ST5 = 0 takt kanalskega tipa RS-232:

ST6	ST7	ST8	kanalska hitrost (baud)
0	0	0	9600
1	0	0	4800
0	1	0	2400
1	1	0	1200
0	0	1	600
1	0	1	300
0	1	1	150
1	1	1	110

13. Začetne nastavive skakačev

Oglejmo si najprej preglednico vseh skakačev, ko imamo tabelo 3.

Oglejmo si nastavitve skakačev pri uporabi epromskih vezij za 4 k zloge, brez čakalnega stanja in pri uporabi diskov 5,25 cole. Najprej imamo:

SK1	SK2	SK3	SK4	SK5	SK6	SK7
0	1	1	0	0	1	0

kjer pomeni 1 stik in 0 prekinitev. Nadalje je:

SK8	SK15A	SK9	SK10	SK11	SK12	SK13	SK14
0	1	0	1	0	1	1	0

SK15	SK16	SK16A	SK17	SK20	SK21
0	1	1	0	0	1

SK18-1	-2	-3	-4	-5	-6	-7	-8	-9
X	0	X	1	0	1	0	0	0

X pomeni 0 ali 1 v odvisnosti od možnosti.

14. Nastavitev frekvence za VCO

S potenciometrom R 14 (list 5) moramo nastaviti nominalno frekvenco 8 MHz za VCO (vezje I 4, list 5). To nastavitev opravimo pri neaktivnem signalu VCOSYNC (ta nastaja v vežju I 21, nožica 24, list 5 in se pošilja v vezje I 8, nožica 5, list 5), ko je vrednost napetosti na nožici 2 vezja I 4, list 5 (to vezje je napetostno krmiljeni oscilator 74 S 124 ali 74 LS 629) približno 3,2 V. Potenciometer R 14, list 5 se v tem stanju nastavi na frekvenco 8 MHz in to

frekvenco merimo na nožici 7 vezja I 4 s frekvenčnim merilnikom.

Predhodno smo ustrezno nastavili tudi skakače in konfiguracijska stikala, kot je bilo opisano.

15. Osnovni V_I sistem (BIOS) za ibmovski mikroračunalnik

15.1. U v o d

Operacijski sistem MS-DOS (Microsoft Disk Operating System) je mogoče namestiti praktično na vsak mikroračunalniški sistem, ki uporablja mikroprocesor 8088 ali 8086. Tudi različni ibmovski operacijski sistemi japonskih, ameriških in evropskih proizvajalcev temeljijo na licenci sistema MS-DOS (vključno s podjetjem IBM). MS-DOS je zgrajen tako, da je del tega sistema dostopen proizvajalcu ali uporabniku mikroračunalniškega sistema in ta del se imenuje BIOS (Basic Input Output System) ali natančneje ROM BIOS (BIOS v ROMu).

IBM je določil svoj ROM BIOS, ki ga je seveda avtorsko zaščitil. S to zaščito naj bi med drugimi zaščitami dosegel, da bi bili zakoniti konkurenčni posnetki računalnikov tipa IBM PC drugačni, manj konkurenčni, predvsem pa čimbolj sistemsko in programsko nezdružljivi z IBM PC. Vendar je s to zaščito IBM dosegel le to, da samo nekateri njegovi storitveni in aplikativni programi niso izvršljivi na konkurenčnih posnetkih, vsi drugi programi in bistveni poslovni in tehnični informacijski sistemi pa se lahko izvajajo na konkurenčnih sistemih dostikrat hitreje in bolj optimalno kot na ibmovem izvirniku.

Pri razvoju ROM BIOSa moramo upoštevati strukturo ibmovskega ROM BIOSa, za katerega imamo zbrane osnovne podatke o programskih prekinitvah (rutinah) v tabeli 4.

15.2. P r o g r a m s k e p r e k i n i t v e t i p a I N T p r o c e s o r j a 8 0 8 8

Ukaza, povezana s programsko prekinitvijo in podatki o njiju so zbrani v tabeli 5.

Ukaz programske prekinitve se lahko uporabi v dva namena, in sicer:

- v programih, ki se testirajo: Ulaz programske prekinitve, ki ima dolžino enega zloga (mnemonika tega ukaza je INT brez operanda, kar ustreza tudi INT 3; njegov strojni kod oziroma zlog je 0cch; za INT 3 pa dobimo iz tabele 5_operacije pri kk = 3 naslov odmičnega dela 12, tj. 0ch in naslov segmentnega dela 14, tj. 0eh), pokliče rutino, katere naslov začenja na lokaciji 0000:000c (glej tabelo 5,

Tabela 4

Prekinitev tipa INT kk kk	Uporaba posameznih subrutin
0	Deljenje z ničlo iz 8088, 8086
1	Enojni korak (pri DEBUG)
2	Nemaskirna parnostna napaka
3	Točka prekinitve (pri DEBUG)
4	Prestop
5	Pisanje na zaslon
6-7	Rezervirano
8-0FH	Prekinitev iz 8259
10H	Video prekinitev, 15 funkcij
11H	Preslikava za naprave
12H	Obseg pomnilnika
13H	Diskovna prekinitev, 16 funkcij za upogljivi in 14 funkcij za trdni disk
14H	Komunikacijska prekinitev, 4 funkcije
15H	Kaseta
16H	Tipkovnična prekinitev, 3 funkcije
17H	Tiskalniška prekinitev, 3 funkcije
18H	Osnovni vstop v ROM
19H	Navezovalni nalagalnik
1AH	Prekinitev časa dneva, 2 funkciji
1BH	Prekinitev zaradi prekinjene tipkovnice
1CH	Časovniška prekinitev (bitje)
1DH	Kazalec v parametričen seznam za video inicializacijo
1EH	Kazalec v seznam diskovnih parametrov
1FH	Kazalec za grafični znakovni vzorec
20H-3FH	Prekinitve DOS
40H	Preusmeritev diskovne prekinitve za sisteme s trdnimi diski
41H	Kazalec v seznam parametrov za trdni disk
42H-FFH	Rezervirano ali uporabljeno za prekinitve DOSa, Basica ali za uporabniške prekinitve

operacije). Ta rutina je navadno del testirnega paketa (DEBUG) in se uporablja za obdelavo točk prekinitve v programih, ki jih testiramo. Iz tabele 4 vidimo, da ta semantika ustreza ibmovskemu ROM BIOSu.

- za klicanje subrutin, katerih naslovi se nahajajo v prvih 1024 zlogih pomnilnika: Pri uporabi dvozložnih ukazov INT kk (kk = 0, 1, 2, ... , 0feh, 0ffh) za programske prekinitve lahko pokličemo katerokoli subrutino od 256 subrutin, katerih naslovi se skladno s shemo CS:IP (kodni segment : ukazni kazalec = 2 zloga : 2 zloga) nahajajo v prvih 1024 zlogih pomnilnika na lokacijah 0, 4, 8, 0ch, 10h, 14h, 18h, 1ch, ... , 0f0h, 0f4h, 0f8h, 0fch.

Ukazi programske prekinitve imajo to dobro lastnost, da uporabijo le en ali dva zloga programskega pomnilnika v primerjavi s petimi zlogi medsegmentnega klica tipa CALL. Razen tega

Tabela 5

mnemo-nika	objektni kod	število zlogov	status I	status T	operacije

INT	1100 110v
	v = 0:
	cch;
	v = 1:
	0cdh;
		1
		
		2
			0	0	(SP):=(SP)-2; ((SP)):=(zastavice);
					(I):=0; T:=0;
					(SP):=(SP)-2; ((SP)):=(CS);
					(SP):=(SP)-2; ((SP)):=(IP);
					(CS):=(vektor(segmentni_del));
					(IP):=(vektor(odmični_del));
					IF v = 0 THEN
					vektor(odmični_del):=(0ch);
					vektor(segmentni_del):=(0eh);
					END IF;
					IF v = 1 THEN
					vektor(odmični_del):=((kk 4)h);
					ktor(segmentni_del):=
					((kk 4+2)h);
					END IF;

IRET	0cfh	1	X	X	(IP):=((SP)); (SP):=(SP)+2; (CS):=((SP)); (SP):=(SP)+2; (zastavice):=((SP)) (SP):=(SP)+2; Vrnitev iz prekinitvene servisne rutine

Tu je (x) vsebina registra x, ((SP)) je vsebina lokacije (SP), ((y)h) je vsebina heksadecimalne lokacije y, kk pa je operand ukaza INT.

se pri teh ukazih avtomatično rešijo v sklad vse zastavice. Klici tipa INT imajo obvezno vrnitev z ukazom IRET (tabela 5, operacije).

Zapišimo zaporedje operacij, ki se sprožijo z ukazom INT:

1. Vsebina zastavic se kopira v sklad.

2. Zastavici IF in TF se anulirata (= 0).

3. Vsebina registra CS se kopira v sklad.

4. Vsebini lokacije na naslovih 00xxx (petmestni heksadecimalni naslov) in 00xxx + 1 se preneseta v register CS (glej sliko 10). Naslov 00xxx je določen z najnižjim (ničtim) bitom operacijskega koda (zloga) ukaza INT ali pa z drugim zlogom ukaza kk (npr. INT kk, kjer je kk drugi zlog). Tu velja:

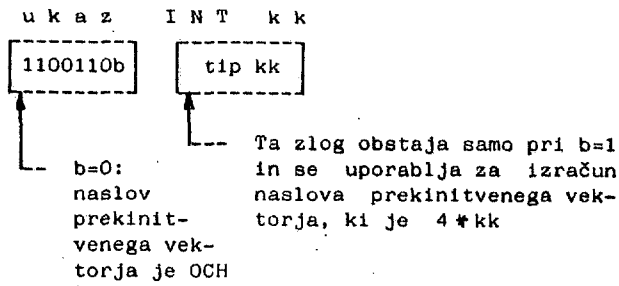
```
IF ničti_bit = 0 THEN
  00xxx := 0eh;
ELSE 00xxx      := (4*kk)+2;
END IF;
```

5. Vsebina registra IP se kopira v sklad.

6. Vsebini lokacij na naslovih 00vvv (petmestni heksadecimalni kod) in 00vvv + 1 se preneseta v register IP (glej sliko 10). Naslov 00vvv je določen z najnižjim (ničtim) bitom operacijskega koda (zloga) ukaza INT ali pa z drugim zlogom ukaza kk (npr. INT kk, kjer je kk drugi zlog). Tu velja:

```
IF ničti_bit = 0 THEN
  00vvv := 0ch;
ELSE 00vvv := 4*kk;
END IF;
```

Imamo tole sliko:



15.3. Navzdolnji razvoj ROM BIOSa

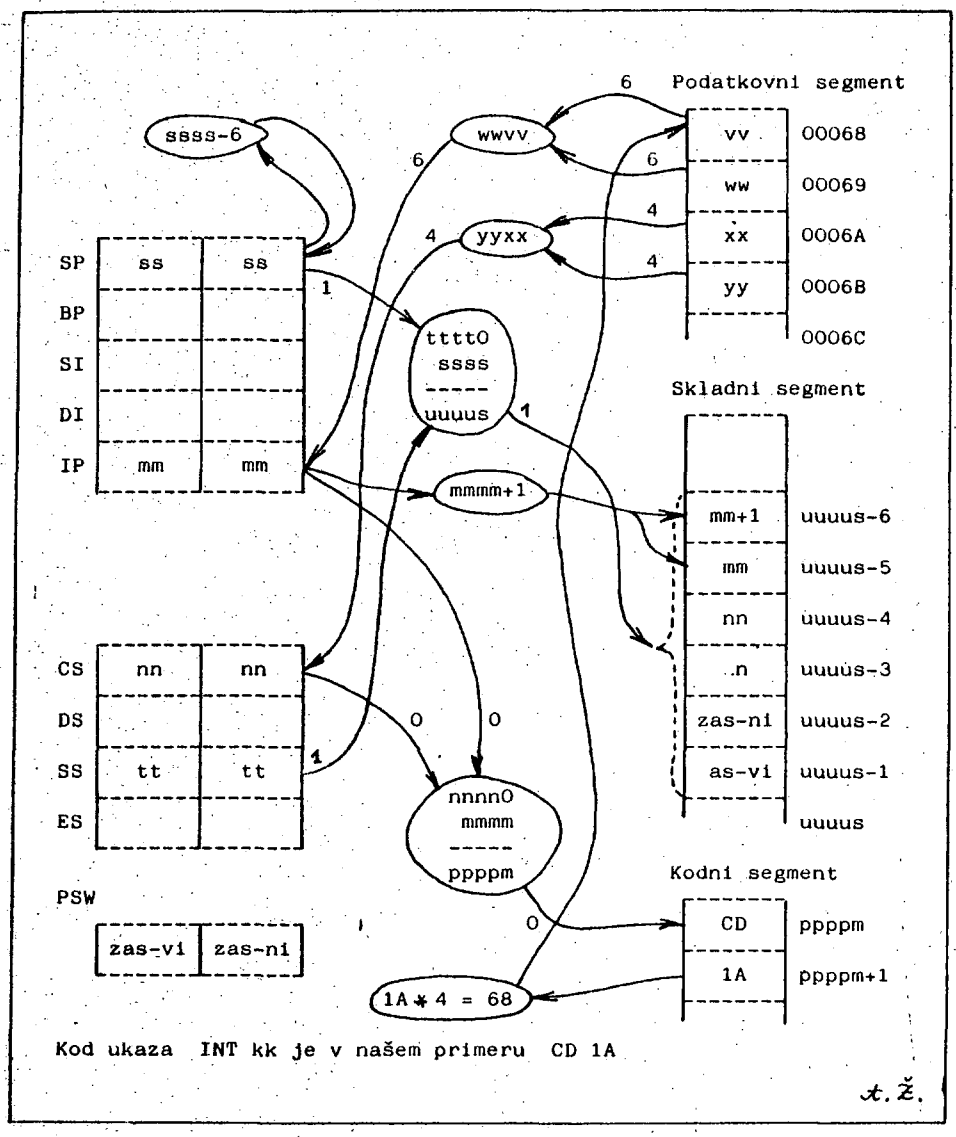
Po ročnem ali vklopnem resetiranju procesorja 8088 imamo

S : IP = 0ffffh : 0000h

To pomeni, da moramo imeti na naslovu 0ffff0h v ROMu prvi ukaz, katerega oblika bo vobče

JMP kodni_segment:odmični_naslov

ali v strojnem kodu z uporabo povratnega zbirnika (zbirka DEBUG.COM)



Slika 10. Ponazoritev delovanja ukaza INT kk

Ta ukaz opravi vrsto operacij: shrani zastavice v sklad, postavi zastavici IF in TF na vrednost 0, naloži CS v sklad, naloži besedo iz ustreznega naslova pomnilnika v register CS, naloži register IP v sklad in vstavi besedo iz ustreznega naslova pomnilnika v register IP.

```
ffff:0000 ea5be000f0      JMP f000:e05b
```

Pri odmičnem naslovu 0e05bh sektorja 0f000h pa bo npr. tale ukaz:

```
f000:e05b ea40e300f0      JMP f000:e340
```

Tako bodo začenjali nadaljnji ukazi ROM BIOSa na lokaciji f000:e340 = 0fe340h.

Napišimo načrt izdelave ROM BIOSa v psevdokodu ((13, 14)), ko imamo navzdolnji razvoj tega programa. Pri tem moramo upoštevati več globalnih elementov, kot so npr. inicializacija vseh perifernih krmilnikov Petre (listi 1, ... , 7 in še posebej Dodatek v ((12)), kjer so zbrani naslovi V_I vrat perifernih krmilnikov Petre), subrutine ibmovskega ROM BIOSa (tabela 4), nalagalnik za naložitev sistemskega nalagalnika (glej kasneje) itn.

Preden začnemo s pisanjem načrta za ROM BIOS, navedimo še nekaj bistvenih ugotovitev o ibmovskemu diskovnemu operacijskemu sistemu (kratko DOS).

15.3.1. Struktura DOS in njegova inicializacija

Ibmovski DOS sestavljajo 4 deli:

1. Nalagalni zapis (sistemski nalagalnik) se nahaja na diskovni stezi 0, v sektorju 1 in na diskovni strani 0 (dvostranska disketa), in to na vsakem disku, ki je bil formatiran z ukazom FORMAT. Nalagalni zapis je na vsakem disku, tako da lahko izda sporočilo o napaki, kadar se želi pognati sistem z disketo, ki ne vsebuje DOS v diskovni enoti A. Pri vinčestrskem disku se ta zapis nahaja v prvem sektorju (sektor 1, glava 0) prvega obroča DOS particije.

2. Povezovalni modul je posebna zbirka (z imenom IOSYS.COM pri Microsoftu ali IBMBIO.COM pri IBM), ki povezuje DOS na spodnji (najnižji) ravnini s perifernimi in drugimi rutinami v ROM BIOSu. Zaradi tega ROM BIOS ne more biti poljuben in mora zadoščati os-

novnim pogojem ibmovskega ROM BIOSa. Povezovalni modul IOSYS.COM je skrita zbirka (ni vidna z ukazom DIR).

3. Operacijski sistem (zbirka MSDOS.COM ali IBMDOS.COM pri IBM), povezuje DOS na zgornji (najvišji) ravnini z uporabniškimi programi. DOS sestavljajo rutine za upravljanje zbirk, podatkovno blokiranje_deblokiranje diskovnih rutin in še vrsta vgrajenih funkcij, v katere je moč enostavno dostopati iz uporabniških programov. Ko se te funkcijske rutine pokličejo iz uporabniških programov, sprejmejo visoko informacijo prek registrov in vsebin krmilnih blokov, nato pa prevedejo te zahteve v enega ali več klicev k IOSYS, kjer se zahteve dopolnijo. MSDOS.COM je skrita zbirka.

4. Zadnji modul je ukazni procesor, ki je odkrita zbirka COMMAND.COM.

To, kar moramo pri razvoju programa ROM BIOS razumeti, je inicializacija DOSa. Ko računalnik Petra zaženemo (z uporabo tipke za resetiranje ali pri vklopu računalnika, ko je v diskovno enoto A vstavljena sistemska disketa), se mora najprej prebrati nalagalni zapis iz steze 0, sektorja 1, strani 0 v hitri pomnilnik (RAM) in ta zapis (nalagalni modul) se mora nato začeti izvajati. Nalagalni modul z diskete pogleda najprej v imenik in preveri, ali sta kot prvi v njem navedeni zbirki IOSYS.COM in MSDOS.COM (natanko v tem zaporedju). Če to ni res, se takoj izda sporočilo o zadevni napaki. Nato se ti dve zbirki prebereta z diska v pomnilnik. IOSYS.COM je prva zbirka v imeniku in njeni sektorji morajo biti strnjeni (ne razpršeni).

Inicializacijski kod zbirke IOSYS.COM opredeli status naprav, resetira diskovni sistem, inicializira priključene naprave, naloži periferne vmesnike in postavi prekinitvene vektorje v nizkem delu pomnilnika; nato pomakne kod prebrane zbirke MSDOS.COM navzdol po pomnilniku in pokliče prvi zlog operacijskega sistema.

Podobno kot IOSYS.COM vsebuje tudi DOS skok v svoj inicializacijski kod; ta kod bo kasneje prekrit s podatkovnim območjem in z ukaznim procesorjem (COMMAND.COM). Tako DOS inicializira svoje notranje delovne tabele, prekinitvene vektorje (za rutine INT 20h do INT 27h) in zgradi prefiks programskega segmenta za zbirko COMMAND.COM v najnižjem prostem segmentu; nato še nastopi vrnitev v kod IOSYS.COM.

Zadnje opravilo inicializacije v programu IOSYS.COM je naložitev zbirke COMMAND.COM na lokacije, ki so bile v ta namen pripravljene z DOS inicializacijo. Program IOSYS.COM je tako svojo nalogo opravil in prenese izvajanje na prvi zlog ukaznega modula COMMAND.

15.3.2. N a č r t R O M B I O S a

S podatki, ki smo jih zbrali o zahtevah združljivosti z IBM PC in na osnovi logičnega vezja mikroračunalnika Petra (listi 1, ... , 7 in dodatek vslovstvu pod navedbo ((12))), lahko začnemo pisati psevdokod za t.i. ROM BIOS.

Osnovna naloga ROM BIOSa bo, da se pri vključitvi računalnika Petra ali ob njegovem resetiranju inicializirajo vsi njegovi V_I krmilniki v odvisnosti od nastavitev konfiguracijskih stikal in da se po tej inicializaciji začne izvajati preprost nalagalnik in ROM BIOSa, katerega naloga je, da iz diskete v diskovni enoti A prebere nalagalni modul (glej prejšnji opis operacijskega sistema) iz steze 0, sektorja 1, strani 0 v RAM in prenese izvajanje na ta modul. Pri tem mora ROM BIOS vsebovati še različne ibmovske subrutine za klice tipa INT 10H do INT 1FH (glej tabelo 4) in ustrezno ibmovsko preslikavo (vektorsko tabelo 6 za te subrutine), ki bo prebrana (pomaknjena) na ustrezno mesto v spodnji del RAMa.

V ROM BIOSu bomo tako imeli ibmovsko tabelo prekinitvenih vektorjev, ki je prikazana absolutno s tabelo 6. Ta tabela se bo preslikala v RAM, začenši z lokacijo 40H. Del RAMa od lokacije 0H navzgor si bomo kasneje lahko izpisali z uporabo storitvenega programa DEBUG.COM in njegoveha ukaza d (d pomeni dump).

Tabela 6 je bila seveda dobljena potem, ko smo v ROM BIOSu že do potankosti sprogramirali vse njene prekinitvene subrutine (tj. subrutine, ki se kličejo z zbirniškim ukazom INT). Programiranje teh subrutin (njihove deklaracije) bomo nakazali tudi kasneje v psevdokodu.

Napišimo si načrt programa v ROM BIOSu v obliki psevdokoda. Imamo tale zapis:

```
PROGRAM rom_bios IS

  INCLUDE začetne_deklaracije;

COMMENT Začetek koda, ki bo rezidenten v
          ROMu;
  Določitev začetke programa (ukaz tipa ORG);

  Onemogočitev prekinitve (ukaz tipa CLI);
  Omogočitev niznih operacij (ukaz tipa CLD);

  Inicializacija segmentnih registrov procesorja 8088;

  Inicializacija perifernih krmilnikov:
    Inicializacija krmilnika 8255;
    Inicializacija krmilnika 8155 (tudi za osveževanje RAMa);
    Inicializacija DMA krmilnika 8237;
    Inicializacija obeh prekinitvenih krmilnikov 8259 (mojstra in pomočnika);
    Inicializacija časovnika 8253;

  Določitev obsega instaliranega pomnilnika tipa RAM;
  Brisanje pomnilnika tipa RAM (vpis 00h na vse lokacije);

  Inicializacija dela prekinitvene vektorske tabele:
    Prekinitve tipa NMI;
    Prekinitev pisanja na zaslon;
    Programske prekinitve;
    Trdne prekinitve;
    Itd.;
```

Tabela 6

Tabelni naslov	Naslov subrutine	Subrutina INT kk	Pomen (ibmovsko združljive prekinitvene subrutine)
f000h:ff13h	f000h:f065h	INT 10h	Video prekinitveni vmesnik
:ff17h	:f84dh	INT 11h	Ibmovske periferne naprave
:ff1bh	:f841h	INT 12h	Preizkus obsega pomnilnika
:ff1fh	:ec59h	INT 13h	Diskovni vhod_izhod
:ff23h	:e739h	INT 14h	Vhod_izhod standarda RS-232
:ff27h	:e5fch	INT 15h	Kasetni vhod_izhod
:ff2bh	:e82eh	INT 16h	Vhod_izhod za tipkovnico
:ff2fh	:efd2h	INT 17h	Vhod_izhod za tiskalnik
:ff33h	:e5fch	INT 18h	Osnovni vstop v ROM
:ff37h	:e6f2h	INT 19h	Navezovalni nalalagalnik
:ff3bh	:fe6eh	INT 1ah	Čas dneva
:ff3fh	:e5fch	INT 1bh	Prepis v RAMu bo: 60h:61h
:ff43h	:e5fch	INT 1ch	Utripanje (bitje) časovnika
:ff47h	:e206h	INT 1dh	Kazalec k video parametru
:ff4bh	:efc7h	INT 1eh	Kazalec k disk. parametrom
:ff4fh	:e5fc	INT 1fh	Kazalec k video razširitvi

```
  Branje stanj konfiguracijskih stikal in na-
    stavitev parametrov;
  Preizkus instaliranosti ibmovskih serijskih
    in paralelnih perifernih naprav;

  CALL navezovalni_nalagalnik;

END PROGRAM rom_bios;
```

V programu rom_bios navedene akcije in subrutine uporabljajo seveda še vrsto drugih subrutin. To so dejansko subrutine programskih prekinitev INT 10h do INT 1fh, ki jih uporablja tudi operacijski sistem. Te subrutine se pravtako nahajajo v rezidentnem delu ROM BIOSa. Opišimo najprej še segment z imenom začetne_deklaracije. Imamo:

```
SEGMENT začetne_deklaracije IS

  Enačbe tipa EQU za naslove vrat perifernih
    krmilnikov (8155, 8255, 8259, 8259, 8253,
    8251, 8251, 8237, 8272A itd.);
  Enačbe tipa EQU za stanja v diskovnih V_I
    rutinah (iztek_časa, operacija_iskanja_neu-
    spešna, krmilnik_8272A_je_slab, CRC_napaka,
    DMA_meja, DMA_prestop, zapis_ni_bil_najden,
    zapisna_zaščita, slaba_naslovna_znamka,
    slab_ukaz, vrata_DMA_strani, vrata_motor-
    ja);
  Deklariranje tipa EXTRN možnih zunanjih ru-
    tin (tistih, ki ne bodo deklarirane v tem
    programu, zaradi možnosti ločenega, modu-
    larnega razvoja oziroma prevajanja);
  Deklariranje tipa EXTRN zunanjih prekinitve-
    nih lokacij (npr.: prekinitveni_vektor_0,
    NMI_vektor; IBM_prekinitveni_vektor, dis-
    kovni_kazalec, kazalec_za_video_parametre,
    itd.);
  Deklariranje začetnega skladovnega segmenta
    (začetek sklada in njegov dopustni obseg);
  Deklariranje rezerviranega podatkovnega obmo-
    čja (zastavica_naprave, obseg_pomnilnika,
    časovniška_beseda, časovniška_zastavica,
    stanje_motorja, število_motorjev, diskov-
    ni_status, diskovna_zastavica, prekinitve-
    na_zastavica, zastavica_realnega_časa, ba-
    za_V_I_vrat, serijska_baza, itd.);

END SEGMENT začetne_deklaracije;
```

V ROM BIOSu imamo še subrutinske deklaracije za rutine iz tabele 6, in sicer v mešanem zaporedju:

```
SUBROUTINE navezovalni_nalagalnik IS

COMMENT To je subrutina INT 19h, ki začenja
  skladno s tabelo 6 na lokaciji 0e6f2 ROM
  BIOSa (psevdoukaz ORG 0e6f2h). Ta rutina je
  enostavni nalagalnik, ki mora prebrati stezo
  0, sektor 1, stran 0 diskovne enote A v pom-
  nilnik tipa RAM, in sicer na lokacijo 0000h:
  7c00h in v to točko se potem prenese tudi iz-
  vajanje programa. Če branje ni uspešno, se
  izvajanje nadaljuje v monitorski točki
  0fc00h: 0000h.;

  Uporabi podatkovni segment BIOSa;
  Inicializacija krmilnika upogljivega diska;
  Bralna zanka:
    Preizkus za 8 ali 5,25 colsko disketo;
    Nastavitev števila sektorjev;
    Branje sektorja 1 steze 0
      (CALL diskovni_V_I);
    Tiskanje sporočila pri napaki in vrnitev v
      monitor;

END SUBROUTINE navezovalni_nalagalnik;


SUBROUTINE čas_dneva IS

COMMENT To je subrutina INT 1ah, ki začenja z
        psevdoukazom ORG 0fe6eh
  Bralni del;
  Pisalni del;

END SUBROUTINE čas_dneva;
```

```
COMMENT  Ostale subrutine;

Definicija tabele programskih prekinitvenih
  vektorjev od INT 10h do INT 1fh;

Definicija trdnih prekinitev (ORG 0e2f0h) od
  INT 40h do INT 4fh;

Definicija diskovne baze, ki jo uporablja sub-
  rutina INT 13h;

Definicija prenosnih hitrosti serijskih
  kanalov;
```

S tem smo končali načrtovanje ROM BIOSa v psevdokodu in tako lahko začnemo s programiranjem v zbirnem jeziku procesorja 8088. Podrobnejši zbirniški programi bodo morda prikazani v drugem prispevku.

16. Sklep k tretjemu delu

Med pisanjem tega nadaljevanja so bile na mikroračunalniku Petra uspešno preizkušene nekatere izvedenke operacijskih sistemov, in sicer: CPM-86, MS-DOS 1.1, MS-DOS 2.0 in nazadnje tudi CCPM. Pri tem so bile uspešno preizkušene tudi nekatere vtične enote za IBM PC (ibmovska tastatura, monokromatični zaslon). Preizkus raznih aplikativnih in sistemskih programskih paketov je v teku (npr. Lotus 1-2-3, Wordstar, Basic, Prolog, Ada, C, makrozbirnik).

Na koncu tega projekta časopisa Informatica bi se želel zahvaliti številnim prijateljem in sodelavcem, ki so s svojo iznajdljivostjo in s svojim delom omogočili uspešno izvedbo projekta.

Slovstvo

((12)) A.P.Železnikar: Ibmovski osebni računalnik Petra II. Informatica 9 (1985), št. 1, str. 9-18.

((13)) A.P.Železnikar: Psevdokodiranje: sintaksa in uporaba. Elektrotehniški vestnik 43 (1976), str. 213.

((14)) A.P.Železnikar: Mikroračunalniški operacijski sistemi. 5 Jugoslavenski seminar o primjeni mikroprocesora MIPRO-82 (zbornik), str. A2-1 do A2-210, Rijeka, 25 - 28 svibnja 1982.

((15)) R.A.King: The MS-DOS Handbook. Sybex, Berkeley 1985.

INFORMATICA '85
61116 Ljubljana, p.p. 2
JUGOSLAVIJA

INFORMATICA 2/85

NEINTEGRISANO OKRUŽENJE PROGRAMSKOG JEZIKA PASCAL

VUKAJLOVIĆ ZLATIBOR

UDK: 519.682.8

ELEKTROTEHNIČKI FAKULTET
SARAJEVO

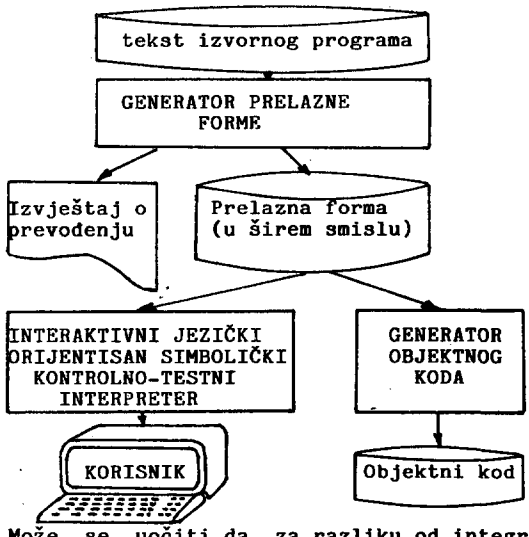
U radu je opsano neintegrisano okruženje za programski jezik Pascal. Ono je ilustrativan primjer svojevrsnog pristupa problematici projektovanja i realizacije prevodilaca, pristupa koji je prvi korak ka projektovanju i realizaciji integrisanog okruženja programskog jezika.

NONINTEGRATED PROGRAMMING LANGUAGE PASCAL ENVIRONMENT: In this paper is described nonintegrated programming language Pascal environment, which is illustrativ example of specific admission to problems of designing and realisation of compilers, as first step on the way of designing and realisation of integrated programming language environment.

1. UVOD

U [1] se obrađuje integrisano okruženje programskog jezika visokog nivoa. Izvjesno je da realizacija takvog okruženja traži znatne resurse, kako sa stanovišta centralne memorije, tako i sa stanovišta eksternih memorijskih uređaja sa mogućnošću direktnog memorijskog pristupa. Za sisteme "siromašnijih" resursa je stoga opravdano (i neophodno) izvršiti dezintegraciju integrisanog okruženja i formirati skup neovisnih programa, koji obavljaju funkcije pojedinih modula integrisanog okruženja.

Takvo jedno neintegrisano okruženje se može predsataviti na način prikazan na slici:



Može se uočiti da, za razliku od integrisanog okruženja programskog jezika, neintegrisano nema mogućnost (modul) inkrementalnog prevođenja, odnosno izmjene prelazne forme. Razlozi za ovo su višestruki:
1. Generator prelazne forme je lakše realizovati od inkrementalnog prevodioca, koji ima mogućnost izmjene prelazne forme, a prilikom njegove realizacije se mogu steći dragocjena iskustva, koja će biti korisna kod realizacije integrisanog programskog okruženja.
2. Slogovi tabele koda se znatno redukuju (sa stanovišta njihove dužine), a djelimično se redukuju i slogovi tabele simbola i tabele teksta, pa stoga datoteke sa ovakvim slogovima zauzimaju znatno manje prostora.
3. Generator objektnog koda i interpreter sa jezički orijentisanim simboličkim kontrolno testnim programom su praktično identični odgovarajućim modulima integrisanog okruženja, pa stoga, trud uložen u njihovu realizaciju neće biti uzaludan. (Potrebno je izvršiti modifikaciju u cilju prilagođenja prelaznoj formi integrisanog okruženja, koja je opštija.)

U ovom radu se opisuje koncept neintegrisanog okruženja za programski jezik Pascal, koji, neosporno, posjeduje znatne kvalitete. Pored opisa prelazne forme u širem smislu, ukratko su komentarisani moduli generatora prelazne forme i generatora objektnog koda, a nešto šire modul interpretera. Za dobro razumjevanje rada, korisno bi bilo da se čitalac upozna sa referencom [1] . Napomenimo, da se pod pojmom pokazivač podrazumjeva referenciranje na neki podatak, a ne pokazivač kao tip (u Pascal-u).

2. OPIS PRELAZNE FORME

Prelaznu formu u širem smislu čine tri tabele: tabela koda, tabela simbola i tabela teksta, a organizovene su kao datoteke sa direktnim pristupom.

Sam program se, u tabeli koda, predstavlja strukturom drveta, u čijim se čvorovima nalaze instrukcije virtualne mašine vrlo visokog nivoa, a koje omogućuju: interpretiranje jezički orijentisanim simboličkim kontrolno testnim programom, sa jedne, i generisanje vrlo efikasnog koda, sa druge strane.

Različiti se tipovi čvorova mogu pojaviti u drvetu. Tako postoji čvor koji odgovara proceduri (tijelu procedure), čvorovi koji odgovaraju pojedinim iskazima jezika (iskaz pridruživanja, iskaz poziva procedure, selektivnom iskazu CASE OF, uslovnom iskazu IF THEN [ELSE],repetitivnim iskazima: WHILE DO, REPEAT UNTIL i FOR, iskazu bezuslovnog grananja GOTO, složenom iskazu, WITH iskazu i lažnom iskazu), čvorovi koji odgovaraju operatorima

svojstvenim jeziku (razni binarni, unarni, funkcionalni i adresni operatori), te čvorovi koji odgovaraju operandima (konstante, varijable i različiti parametri).

U čvoru koji odgovara tijelu, od strane programera definisanih procedura (funkcija), nalaze se podaci o dužinama varijabli i parametara, pokazivači na ugnježdene procedure (funkcije) i procedure (funkcije) istog statičkog nivoa ugnježdenja, te pokazivač na odgovarajuće iskaze, kao i pokazivač na tabelu simbola.

Čvorovi koji odgovaraju iskazima jezika sadrže podatke: o tipu iskaza, pokazivač na tabelu teksta (gdje se nalazi tekst koji odgovara datom iskazu), kao i pokazivač na slijedeći iskaz istog statičkog nivoa ugnježdenja. Ostali podaci zavise od tipa iskaza, a to su (sumarno): pokazivači na ugnježdene iskaze i različite operande (konstante, varijable, parametre), statički nivoi ugnježdenja pozivajuće i pozvane procedure i sl.

Čvorovi unarnih i binarnih operatora sadrže podatke o operaciji i reference na odgovarajuće operande, a čvorovi koji odgovaraju funkcionalnim operatorima: pokazivač na parametre, statičke nivoe ugnježdenja pozivajuće i pozvane procedure i pokazivač na tijelo funkcije. Adresni operatori (indeksiranja, pomaka i indirekcije), pored pokazivača koji omogućava izračunavanje bazične adrese, sadrži i podatke (zavisno od tipa) o donjoj i gornjoj granici indeksa, pokazivač na poddrvo kojim se vrijednost izraza izračunava, dužinu elementa niza, odnosno relativan položaj polja u odnosu na početak sloga.

Čvorovi operanada sadrže podatke o vrsti (konstanta, varijabla, različiti parametri) i tipu operanda, te podatke o vrijednosti (konstanti), odnosno podatke koji omogućuje izračunavanje adrese (varijabli i različitih parametara). Date informacije su, ustvari, sadržane i u tabeli simbola, a ovdje se multipliciraju zbog toga što se očekuje da će im se, u tom slučaju, brže pristupati. Naime, slogovi tabele koda su duplo kraći, a velika je vjerovatnoća da će se nalaziti u bloku u kojem su i čvorovi iz kojih se vrši referenciranje na njih.

Pored nabrojanih, postoje i čvorovi koji odgovaraju gornjoj (donjoj) granici FOR TO (DOWNTO) iskaza, čvorovi odgovarajućih labela CASE iskaza, čvorovi različitih parametara (uz pozive procedura,odnosno funkcija), čvor konstruktora skupa, te čvorovi različitih konstanti koje nisu skalarnog tipa.

Izuzev činjenice da su joj, zbog direktnog pristupa, slogovi fiksne dužine, u organizaciji tabele simbola nema posebnih specifičnosti.

Tabela teksta je takođe datoteka sa direktnim pristupom i sa slogovima fiksne dužine. Svaki slog sadrži podatke o dužini teksta koji odgovara iskazu iz kojeg se vrši referenciranje na dati slog, broju vodećih blanko znakova i sam tekst. Ukoliko je tekst duži od sloga, nastavlja se u slijedećem slogu tabele teksta.

3. OPIS MODULA NEINTEGRISANOG OKRUŽENJA

Od već spomenuta tri modula neintegrisanog okruženja, svakako je najkompleksniji i za realizaciju najteži modul generatora prelazne forme. To je ustvari prevodilac izabranog programskog jezika (Pascal), koji generiše kod virtualne mašine (čije su instrukcije - čvorovi napred opisani). Dodatni zahtjev na prevodilac je zahtjev za formiranje tabele teksta, odnosno "logičnih" linija, za svaki iskaz iz programa po jednu ili više. Ovaj modul se, kao i ostali, može realizovati u jeziku visokog nivoa (npr. u Pascal-u).

Modul jezički orijentisanog simboličkog kontrolno testnog programa interpretira generisani kod virtualne mašine koristeći odredeni skup procedura i fuknkcija. Podaci nad kojima se operiše, nalaze se na steku, koji se definiše kao niz karaktera po imenu STACK.

Za dokučivanje sa steka podataka različitih skalarnih i realnog tipa postoji skup funkcija, koje za konstantni parametar imaju početnu adresu , odnosno vrijednost indeksa iz tabele STACK, a rezultat funkcije je odgovarajući podatak. U realizaciji datih funkcija se koristi varijabilan slog. Npr., ako cjelobrojni podatak ima dužinu identičnu dužini dvaju karaktera, odgovarajuća funkcija ima oblik:

```
FUNCTION integerfetch(a:integer):integer;
VAR x:RECORD
      CASE karakterni_oblik:boolean OF
       true:  x11,x12:char;
       false: x2:integer
      END
     END;
BEGIN x.karakterni_oblik:=true;
x.x11:=STACK a ;x.x12:=STACK a+1 ;
x.karakterni_oblik:=false;
integerfetch:=x.x2
END;
```

Za smještanje podataka različitih skalarnih i realnog tipa, postoji skup odgovarajućih procedura, koje imaju dva konstanta parametra: sam podatak i adresu na kojeg se isti smješta, a realizuju se na način sličan realizaciji funkcija za dokučivanje podataka sa steka.

Za izračunavanje adrese nekog objekta (varijable, parametra ili rezultata funkcije), koristi se procedura čije zaglavlje ima oblik:

```
FUNCTION address(x:integer):integer;
```

Ova funkcija izračunava adresu objekta, čiji je opis dat u čvoru operanda na kojeg x pokazuje, a rezultat funkcije je odgovarajuća adresa (indeks iz tabele STACK).

Za izračunavanje vrijednosti izraza odgovarajućeg tipa, koristi se odgovarajuća iz skupa funkcija za aritmetiku. Npr. za cjelobrojnu aritmetiku, zaglavlje funkcije ima oblik:

```
FUNCTION intval(x,y:integer):integer;
```

Konstantni parametar označava način na koji se izraz izračunava, a naznačuje da li je izraz konstanta, neki drugi operand ili izraz predstavljen drvetom u čijem se korijenu nalazi binarni, unarni ili funkcionalni operator. Drugi parametar označava ili samu konstantu ili pokazivač na odgovarajući čvor tabele koda. Data funkcija može rekurzivno pozivati samu sebe, funkciju za dokučivanje (odgovarajućeg) operanda sa steka ili pak aktivirati funkcije definisane od strane programera (koje su dio tabele koda).

Kod realizacije iskaza pridruživanja, koji operiše nad objektima komponentnog tipa, koristi se procedura čije je zaglavlje oblika:

```
PROCEDURE moveblock(x,y,z:integer);
```

gdje x i y označava izvornu, odnosno odredišnu adresu, a z dužinu bloka podataka. Slično datoj, postoji i procedura koja na odredišnu adresu smješta konstanu zapisanu u tabeli koda.

Za ispis teksta izvornog programa odgovarajućeg iskaza, koristi se procedura čije je zaglavlje oblika:

```
PROCEDURE line(x:integer);
```

gdje je x pokazivač u tabelu teksta. Ova

procedura se poziva na početku interpretiranja svakog iskaza, a ispis se vrši samo ako je izabran odgovarajući način rada .1.,odnosno ako je data linija označena kao trag ili prekidna tačka.

Za ispisivanje rezultata izračunavanja skalarnog ili realnog tipa pojedinih iskaza, koristi se odgovarajuća iz skupa procedura. Npr. zaglavlje procedure za ispis podatka bulovog tipa ima oblik:

```
 PROCEDURE writebool(x:boolean);
```

Ispis se vrši ako je izabran koračni način rada ili ako je izabran iskazni način rada i ako pokazivač na zaglavlje skupa lokalnih objekata ima odgovarajuću vrijednost (zavisno od toga da li se koristi rastuća ili opadajuća varijanta steka). Za ispis promjenjivih (konstanti) čiji tip nije predefinisan (predefinisani su npr. integer, real, char, ...), odgovarajuća procedura sadrži, kao konstantne parametre adresu podatka i pokazivač na opis tipa u tabeli simbola, što omogućuje da se isti ispiše u simboličkom obliku. Nakon ispisa rezultata izračunavanja, proces interpretiranja se prekida. Interpretiranje se nastavlja zadavanjem komande kojom se definiše novi način rada.

Najzad, tu je i procedura koja interpretira pojedine iskaze, odnosno procedura čije je zaglavlje oblika:

```
 PROCEDURE code(p,pl,sl:integer);
```

gdje p pokazuje na odgovarajući čvor iskaza, dok su pl i sl statički nivoi ugnježndenja procedure, odnosno iskaza (unutar procedure).Poslednja dva se koriste da bi se, nakon bezuslovnog grananja na iskaz nižeg statičkog nivoa ugnježdenja unutar date procedure, odnosno prvi nivo statičkog nivoa ugrnježdenja kod grananja izvan iste, mogla korektno prebaciti kontrola izvršenja iskaza. Oni, takode, omogućiju formatizirani ispis teksta izvornog programa.

"Kostur" procedure za interpretaciju iskaza ima oblik:

```
VAR cvor:element_tabele_koda;
   ,..

BEGIN
WHILE p0 DO
 IF lift
  THEN
   BEGIN
    IF (plgotopl)OR(plgoto=pl)
                  AND(slslgoto)
    THEN p:=0
    ELSE
     BEGIN lift:=false;p:=PGOTO END
   END
  ELSE
   BEGIN
    ( Ako je zadana nova komanda,
        interpretiraj je;
       Dokuči p-ti element tabele
        koda i smjesti ga
        u promjenjivu cvor;
       Pozovi proceduru za ispis
        linije (sa odgovarajućim
        parametrom);
       Interpretiraj dati čvor
         na odgovarajući način    )

         , . .

         p:=cvor.slijedeći_iskaz
   END
END;
```

Lift je varijabla koja se na vrijednost "true" postavlja u procesu izvršenja GOTO iskaza, kada se definišu i vrijednosti varijabli plgoto, slgoto i PGOTO, a na osnovu podataka sadržanih u instrukciji grananja. Ostali dio "kostura" je, zbog pojednostavljenja, samo opisno definisan.

Kod interpretacije nekih nekih čvorova (iskaza) jezika, procedura code rekurzivno poziva samu sebe.. Npr. čvor koji odgovara REPEAT UNTIL iskazu, interpretira se na način:

```
REPEAT
( predhodno je ispisan tekst iskaza )
code(cvor.ost0,pl,sl+1)
 ( u cvor.ost0 je pokazivač
       na ugnježdene iskaze )
line(cvor.ost1) (linija sa UNTIL );
b:=boolval(cvor.ost2,cvor.ost3)
 ( b je pomoćna varijabla, a pomoću
    cvor.ost2 i cvor.ost3 se izračunava
    vrijednost uslova     );
writebool(b)
UNTIL b;
```

Dakle iskaz se interpretira pomoću samog sebe.

Na sličan način se interpretiraju standardne procedure i funkcije, a i ostali iskazi.

Iz kostura se može uočiti da se, kod interpretacije svakog iskaza provjerava da li je zadana nova komanda, te se, ako jeste, ista i interpretira. To omogućuje da se u bilo kojem trenutku testiranja programa izvršavanje u tragovnom ili normalnom načinu može prekinuti, i nastaviti sa načinom definisanim novom komandom.

Da bi se mogao istovremeno izvršavati ispis na ekran i čitati znak sa tastature (unositi komande) terminal mora da radi u punom dupleks načinu , a što operativni sistemi računara dozvoljavaju. Medutim, ovaj zahtjev, kao i potreba za datotekama sa direktnim pristupom, može dovesti do toga, da se, jedan manji dio koda, piše u nekom drugom (prvenstveno asemblerskom) jeziku.

4. ZAKLJUČAK

Neintegrisano okruženje programskog jezika ima za cilj uvodenje jedne (nove) prakse u projektovanju i realizaciji prevodilaca. Ovakav pristup ima svoje odredene prednosti, najviše sa stanovišta prenosivosti i udobnosti testiranja programa. Moduli generatora prelazne forme i interpretera su veoma prenosivi. Modifikacije je potrebno izvršiti zbog različitih dužina predefinisanih tipova na pojedinim računarima, a što ne izaziva neke bitne probleme. Nešto veći problemi nastaju objektnog koda, pri čemu će iskustva stečena na jednom računaru, biti veoma korisna kod realizacije na drugom. Bitno je napomenuti, da struktura drveta izabrana za predstavljanje tabele koda omogućuje generisanje vrlo efikasnog (objektnog) koda. Nadalje, prilikom pisanja prevodioca, programer neće biti opterećen različitim detaljima ciljne mašine. Konačno, neintegrisano okruženje može biti realizovano od strane više ljudi, čiji su poslovi praktično nezavisni.

Interpretiranje u procesu testiranja, čiji je nedostatak relativno mala brzina izvršenja, ma koliko sporo, opravdano je iz razloga što brzina izvršenja (kod testiranja) nije presudan faktor.

5. LITERATURA:

Z. Vukajlović: Integrisano okruženje programskog jezika - alat za udobno i efikasno programiranje

INFORMATICA 2/85

ANALIZA RASPOREDJIVANJA ZADATAKA U MULTIPROGRAMSKOM SISTEMU (MPS-u) KORIŠTENJEM SIMULACIJE U PASKALU

GOJO ŠTRKIČ,

DAVORIN NOVOSEL

ELEKTROTEHNIČKI FAKULTET
BANJALUKA

UDK: 681.3.012

SADRŽAJ - U radu je definisan problem rasporedjivanja u MPS-u, Zatim se uz elementarne termine iz simulacije date osnovne osobine simulacionih jezika kao i način kako su iste riješene u paskalu. Dat je model MPS-a i na osnovu njega su u paskalu simulirana tri, po kompleksnosti različita, algoritma. Rezultati simulacije ukazuju da na performanse sistema utiču kako način dodjeljivanja prioriteta tako i način podjele resursa.

ABSTRACT - In this paper is defined problem of scheduling in multiprogrammed system (MPS). Beside elementary simulation terms here are given basic features of simulation languages as well as ways of solution these features in pascal.Model of MPS is given and in pascal are simulated three, in complexity different, algorithms based on the model. Results of the symulation show that on performances of MPS influence have both way priority assingment and way of assignment of resources.

1. UVOD

Multiprogramski sistem karakteriše konkurentno izvodjenje više zadataka. Kada broj zadataka prijedje odredjeni limit onda se selektuju samo oni zadaci koji mogu nastaviti izvodjenje a ostali moraju čekati. Operaciju selektovanja zadataka (scheduling) u operacionom sistemu izvodi poseban modul koji se zove rasporedjivač (scheduler).

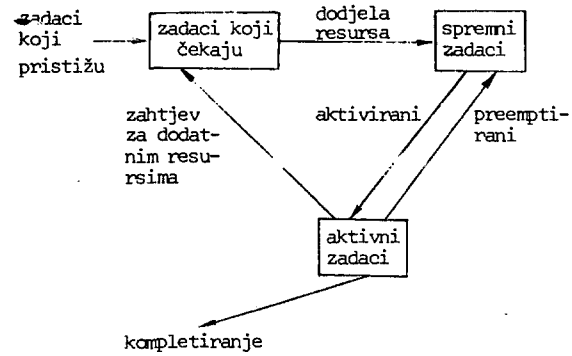
Poznato je da se koristeći opšti algoritam baziran na prioritetima zadataka mogu implementirati različite strategije rasporedjivanja |4,5|. Pored toga rasporedjivanje se može podijeliti u dvije posebne funkcije: funkciju dodjeljivanja prioriteta i funkciju dodjeljivanja resursa.

Mi smo koristili simulacione modele kako bi razmatrali osobine algoritama rasporedjivača. Prilikom izbora jezika u kom bi izveli simulaciju imali smo u vidu da se za simulaciju može koristiti bilo koji jezik opšte namjene a postoje i posebni, simulacioni jezici, u koje su ugradjene osobine za prikupljanje statistike kao i za manipulaciju diskretnim dogadjajima.

U literaturi postoji niz analiza (na pr. |2|) u kojima je imajući u vidu neke parametre kao što su podesnost za simulaciju, fleksibilnost, zahtjevi za memorijskim prostorom i procesorskim vremenom, poredjeno više jezika kako simulacionih tako i jezika opšte namjene. Ishod ovih analiza je pokazao da su, ako se izuzmu simulacioni jezici (koji takodje imaju svoje nedostatke), strukture paskala dovoljne za simulaciju računarskih sistema.

2. UOPŠTE O MPS-u I MODELU MPS-a

Usvojićemo model računarskog sistema onako kako je to prikazano na sl.1.



Sl.1. Stanje zadataka

Rad sistema opisan ovim modelom može se objasniti tokom zadataka. Zadaci koji pristižu mogu zahtijevati isključiv pristup na resurse kao što su magnetske trake i diskovi. U slučaju da ovi resursi nisu raspoloživi dati zadaci moraju čekati. Kad resursi postanu raspoloživi zadaci postaju spremni za aktiviranje. Mi smo usvojili da zadatak postaje aktivan kad mu se dodijeli virtualni processor i potreban blok memorije.

Razmatrali smo i mogućnost da aktivni zadatak može biti preemptiran iz aktivnog stanja čime on oslobadja svoj blok glavne memorije i svoj virtualni procesor i ponovo se pridružuje skupu spremnih zadataka. Preemptirani zadaci mogu biti reaktivirani i nastaviti izvodjenje od tačke preemptiranja na bilo kom procesoru i u bilo kom bloku glavne memorije. Isto tako, zadatak može kompletirati izvodjenje i završiti. Analizirana je i mogućnost da aktivni zadaci mogu zahtijevati dodatne resurse (na pr.

dodatnu memoriju). Ako ovakav sistem podržava preemptivno rasporedjivanje tada dati zadaci mogu biti preemptirani u stanje čekanja ili spremno stanje zavisno od raspoloživosti zahtijevanih resursa (vidjeti sl.1.).

Sve tri algoritma koje smo razmatrali predpostavljaju da se zadaci nalaze na listama koje su poredane po prioritetima. Zadatak rasporedjivača je da pretraži listu spremnih zadataka i selektuje za aktiviranje neke od zadataka. Zavisno od raspoloživih resura, rasporedjivač ponekad preemptira aktivne zadatke i vraća ih u spremno stanje.

Analiza je vršena na osnovu nekih pojmova koji zahtijevaju objašnjenje. Naime, ako zadatak stiže u trenutku t_ip a završava se (odlazi) u trenutku to tada je $t_ic - t_ip$ vrijeme protoka datog zadatka. Za skup zadataka $\{Z_i\}$, i=1,...,n; sa vremenima pristizanja tp i vremenima odlaska to, vrijeme završavanja za {Zi} je $\max_{1<n}(t_io)$ a prosječno vrijeme protoka je

$$\frac{1}{n}\sum_{i=1}^{n}(t_io - t_ip) \quad .$$

3. STRATEGIJE RASPOREDJIVANJA I PRIORITETI

Prioriteti se mogu dodijeliti izvana a postoji i alternativa da sistem sračunava prioritete na osnovu nekih atributa zadataka kao što su trenutni zahtjevi za memorijom, predvidjeno vrijeme procesiranja, vrijeme pristizanja i slično. Mi smo vršili analizu na osnovu slijedećih strategija rasporedjivanja.

Korištene strategije rasporedjivanja

strategija		željeni efekat
NkVP	Najkraće vrijeme procesiranja	Minimizira vrij.procesiranja
NdVP	Najduže vrijeme procesiranja	Smanjuje ukupno vrijeme
NmZM	Najmanji zahtjevi memorije	Zadaci se izvode po važnosti
	Vanjski prioriteti	

4. SIMULACIJA U PASKALU

Prvo ćemo uz osnovne osobine simulacionih jezika dati i elementarna objašnjenja nekih pojmova koji se koriste u simulaciji, a zatim ćemo u tom svjetlu iskomentarisati odgovarajuće osobine paskala.

U literaturi se osnovni element modula sistema često naziva entitet a osobine entiteta se nazivaju atributima. Simulacioni jezici posjeduju mogućnosti da se pomoću njih mogu opisivati entiteti i njihovi atributi.

S obzirom da entiteti ne mogu biti opsluženi u momentu kad se pojave to simulacioni jezik mora imati mogućnost za manipulacijom repovima (listama) čekanja.

Svaki simulacioni jezik ima mogućnost da se pomoću njega u simulator sistema može ugraditi simulacioni sat na kom se vrijeme uvećava nakon svake promjene stanja sistema.

Ako se prihvati da su dogadjaji promjene u stanju sistema onda je aa očekivati da svaki simulacioni jezik posjeduje mogućnost za opis i rasporedjivanje dogadjaja.

Simulacioni jezici imaju mogućnost za prikupljanje statistike generisane simulacijom.

Kod simulacije računarskih sistema uobičajeno je da se radno opterećenje računara opisuje stacionarnim distribucijama vjerovatnoće bez obzira na to koji se programski jezik koristi kao sredstvo za simulaciju.

Mi smo u tu svrhu koristili generator slučajnih brojeva (uniformna raspodjela) koji ima na računaru VAX 11/780 VMS.

Gledano sa stanovišta simulacije, ono što paskal čini podesnim je pokazivački tip podataka (pointer type) i odgovarajuća dinamička varijabla. Na pr. iskaz

```
type Tp = ↑T
```

ukazuje da su vrijednosti pokazivačkog tipa T_p ustvari pokazivači za podatke tipa T. Uz pokazivački tip idu i dvije standardne procedure new() i dispose() za koje ćemo dati slijedeće objašnjenje.

Ako je P pokazivačka varijabla tipa T_p onda procedura new (P) zapravo alocira varijablu tipa T, generiše pokazivački tip T_p koji referencira novu varijablu i pridružuje mu vrijednost P. Inverzna procedura od new() je dispose() koja oslobadja memorijski prostor zauzet odgovarajućom dinamičkom varijablom.

Sada ćemo ilustrovati kako se atributi entiteta opisuju pomoću paskala. Na sličan način je definisan zadatak i njegovi atributi u podnaslovu 5. Naime, pokazivač zadatka pokazuje na varijablu zadatak koja je tipa zadatak _rec. atributi varijable zadatak specificiraju se pomoću polja tipa RECORD.

```
TYPE zadatak =↑zadatak_rec
     zadatak_rec = RECORD
                    pr: integer;
                    sljedeći: zadatak
                   END;

VAR proces: zadatak;
```

Kreiranje zadatka vrši se iskazom

```
new(proces);
```

a dodjela atributa može se vršiti na slijedeći način:

```
proces↑.prioritet  := 10;
```

Manipulaciju listama čekanja ilustrovati ćemo pomoću procedure INSERT koja prema prioritetu ubacuje zadatak na listu spremnih zadataka. Ova procedura može poslužiti kao primjer kako se, na odgovarajuću listu, mogu ubacivati entiteti prema veličini bilo kog atributa.

```
procedure insert (VAR nZ,lp : zadatak);
var ubacen: boolean;
    pom,pomslj: zadatak;
begin
IF lp=nil THEN
 (* lista prazna *)
  begin nZ↑.sljedeći:=lp; lp:=nZ end
ELSE BEGIN
if lp↑.pr < nZ↑.pr then
 (* prioritet veći od prioriteta čela liste *)
  begin nZ↑.sljedeći:=lp; lp:=nZ end
else begin
(* prioritet manji od prioriteta čela liste *)
 pom:=lp;
 ubacen :=false;
 while (ubacen=false) and (pom<>nil) do
  begin
      pomslj:=pom↑.sljedeći;
     IF pomslj = nil THEN
       BEGIN
        nZ↑.sljedeći :=nil; pom↑.sljedeći:= nZ;
        ubačen:=true;
        (* lista ima 1 elem. pa je kao manji ubačen
           iza njega *)
        END
      ELSE
      BEGIN
      if (nZ↑.pr <= pom↑.pr) and (nZ↑.pr > pomslj↑
         .pr) then
      begin
        nZ↑.sljedeći:=pom↑.sljedeći;
        pom↑.sljedeći:=nZ;
        (* zadatak ubačen na listu *)
        ubačen = true
        end
        else
        begin pom := pom↑.sljedeći end
        (* pomijera se na sljedeći element *)
        END
    end
END
END;
```

5. RASPOREDJIVANJE ZADATAKA

Sada ćemo opisati tri algoritma za selekciju zadataka i razmotriti implikacije koje nastaju povećavanjem kompleksnosti algoritama. Za detaljan uvid u algoritme pogledati podnaslov 8. U podnaslovu 3. je spomenuto više korištenih strategija raspoređivanja s tim da je jedino data procedura za ubacivanje na listu prema vrijednosti prioriteta dodijeljenih izvana. Na isti način se vrši dodavanje na odgovarajuću listu prema veličini bilo kog atributa. Uzimanje sa liste vrši se sa čela bez obzira kako je na listu ubacivano.

Svaki od algoritama mora imati slijedeće osobine:

1. Aktiviranje zadatka najvišeg prioriteta koji će se zadovoljiti raspoloživim resursima.

2. Osiguranje efikasnog korištenja sistemskih resursa (virtualnih procesora i glavne memorije).

U sva tri algoritma aktivni i spremni zadaci su na listi poredani od najvišeg do najnižeg prioriteta.

Proces raspoređivanja se starta kadgod se stanje zadatka promijeni, tj. kad zadatak ulazi u stanje spreman ili kad se zadatak završi.

U algoritmima je korišteno nekoliko pomoćnih procedura čiju ćemo funkciju objasniti. Procedura AKTIVIRAJ (zad, VP) aktivira zadatak zad na virtualnom procesoru VP. Procedura Preemptiraj (zad, VP) deaktivira zadatak zad na virtualnom procesoru VP.

Šta se dešava prilikom pojedinih događaja prikazano je na slijedećem pregledu na sl.2., gdje je uz događaj data i odgovarajuća aktivnost.

1. Zadatak zad (prvi put) ulazi u stanje spreman

```
zad↑.prioritet:= ... ;  (* dodjela prioriteta *)
zad↑.Zaht_mem:=... ;    (* dodjela memorije *)
zad↑.Zadatak_status:=spreman_Z;  (* postavljanje statusa *)
insert (zad, lista_spremnih_
        zadataka) ;              (* ubacivanje na
                                    listu spremnih *)
INICIJALIZIRAJ (Raspoređivač) ;
```

2. Zadatak zad postaje aktivan na virtualnom procesoru VP

```
zad↑.zadatak_status:=aktivan_Z;
VP↑.VP_status :=preemptivan_VP; (* ako procesor
                                   može preuzeti
                                   zadatak višeg
                                   prioriteta *)
ili VP.VP_status :=nepreemptivan_VP; (* u suprotnom *) .
```

3. Zadatak zad završava preemptiranje virtualnog procesora VP

```
zad↑.zadatak_status:=spreman_Z;
(* Uvećaj broj VP za jedan *)
Raspoloživa_mem :=Raspoloživa_mem + zadatak↑.
                  Zaht_mem;
VP↑.status :=slobodan_VP ;
(* vrati procesor u skup nezauzetih VP *)
INICIJALIZIRAJ (Raspoređivač) ;
```

4. Zadatak zad se završava na virtualnom procesoru VP

```
zad↑.status  :=kompletiran_Z ;
(* Uvećaj broj VP za jedan *)
Raspoloživa_mem :=Raspoloživa_mem + zadatak↑.
                  Zaht_mem;
(* izbaci zadatak zad iz liste aktivnih (spremnih) zadataka *)
VP↑.status := slobodan_VP
(* ubaci VP u listu nezauzetih procesora *)
INICIJALIZIRAJ (Raspoređivač);
```

Sl.2. (Vidjeti podnaslov 8.)

Algoritam 1 ima osobinu da aktivira spremne zadatke i to od najvišeg prioriteta do najnižeg.

Karakteristika mu je da vrlo slabo iskorištava resurse sistema. Kao najjednostavniji rasporedjivač zadataka on pretražuje listu spremnih zadataka i dodjeljuje zadatke virtualnim procesorima sve dok ne naidje na prvi zadatak koji nije spreman.

Samo se po sebi nameće da bi rasporedjivač bio bolji kada bi pretraživao čitavu listu tražeći zadatke koji se zadovoljavaju sa raspoloživom memorijom. To je postignuto Algoritmom 2. Njegove karakteristike u poredjenju sa algoritmom 1 su:

- povećan rizik da će zadatak biti rasporedjen van strogog držanja prioriteta,
- povećana potencijalna efikasnost rasporedjivanja.

Poboljšanje performansi Algoritma 2 može se postići uvodjenjem preemptivnih osobina koje dozvoljavaju aktivnim zadacima da budu deaktivirani i vraćeni u listu spremnih zadataka. Ovi preemptirani zadaci mogu biti reaktivirani na bilo kom virtualnom procesoru i u bilo kojoj tački u glavnoj memoriji. Intencije Algoritma 3 su:

- da drži aktivnim zadatak sa najvišim prioritetom koji će zadovoljiti ograničenja resursa,
- da omogući da zadaci nižeg prioriteta budu preemptirani kako bi, u slučaju potrebe, oslobodili resurse za zadatke višeg prioriteta.

Pored liste spremnih i aktivnih zadataka postoji lista aktivnih virtualnih procesora poredana u obrnutom redoslijedu prioriteta pripadnih aktivnih zadataka. Počev od čela ove liste vrši se pretraživanje da bi se našli zadaci koje treba preemptirati. Stanje procesora na ovoj listi označava se sa preemptivan_VP ako je procesor preemptivan ili nepreemptivan_VP u suprotnom. Status nepreemptivan_VP se koristi i u slučaju kad procesor mijenja stanje kako bi mogao bez prekida izvesti operaciju mijenjanja stanja. U algoritmu figuriraju i varijable Br_VP_raspol_za_raspor i Mem_raspol_za_raspor, koje označavaju raspoloživi broj virtualnih procesora i veličinu raspoložive memorije respektivno. Treba imati u vidu da u vrijednosti ovih varijabli ulaze i oni resursi koji su u posjedu nekih zadataka s tim da su ovi zadaci preemptivni. Rasporedjivač pretražuje listu aktivnih i spremnih zadataka počev od čela tj. zadatka sa najvišim prioritetom pa naniže. Pri tome mogu nastati slijedeći slučajevi:

- Ako je zadatak već aktivan a zadovoljiće se raspoloživim resursima on ostaje aktivan a ažuriraju se brojači resursa.

- Ako je zadatak spreman a zadovoljiće se trenutno raspoloživim resursima, ažuriraju se brojači resursa i starta procedura AKTIVIRAJ.

- Ako je zadatak spreman a zadovoljiće se raspoloživim resursima ali ne trenutno raspoloživim onda postoji zadatak nižeg prioriteta koji će biti preemptiran kako bi napravio prostora datom zadatku. U tom slučaju se ažuriraju dvije varijable Zaht_br_VP i Sum_zaht_mem kako bi se u sljedećoj fazi odredio broj VP i veličina memorije koje treba preemtirati. Operacija preemptiranja se izvodi tako što se pretražuje lista kativnih procesora poredana prema opadajućem prioritetu pripadnih zadataka. Pretraživanje se prekida kad se obezbijedi dovoljno resursa. Sa preem_Z_u_toku se označava status zadataka za koga je preemtiranje u toku, a odgovarajući resursi koji će kasnije stati na raspolaganje ažuriraju se pomoću varijabli VP_za_koje_je_preem_u_toku i Mem_za_koju_je_preem_u_toku.

6. REZULTATI SIMULACIJE

Vrijednosti ulaznih parametara korištenih u simulaciji:

parametar	minimum	maksimum
zahtjevi memorije	10 jedinica	40 jedinica
prioritet	1	100
veličina memorije	200 jedinica	200 jedinica
broj virt.proc.	10	10

Srednje vrijeme protoka i ukupno vrijeme protoka dato prema strategijama rasporedjivanja i algoritmima:

	Algoritam 1		Algoritam 2		Algoritam 3	
	vrijed.	strateg.	vrijed.	strateg.	vrijed.	strateg.
SVP min.	117	NkVP	106	NkVP	91	NkVP
maks.	301	NdVP	209	NdVP	189	NdVP
UV min.	430	NmZM	390	NdVP	382	NdVP
maks.	461	NkVP	415	NkVP	430	NkVP

SVP - srednja vrijeme protoka
UV - ukupno vrijeme

Vrijednosti date u gornjoj tabeli mogu varirati od mašine do mašine kao i od opterećenja sistema ali njihov medjusobni odnos ostaje uglavnom stalan.

7. ZAKLJUČAK

Prezentirana su tri različita algoritma rasporedjivanja i dati su rezultati analize korištenjem simulacionog modela pisanog u paskalu. Rezultati analize ukazuju na to da na performanse sistema utiče kako način dodjeljivanja prioriteta tako i način dodjeljivanja resursa. Pored toga prezentiran je paskal kao orudje za simulaciju. Izdvojene su osobine simulacionih jezika i imajući iste u vidu prezentirane odgovarajuće osobine paskala pomoću kojih je napravljen simulacioni model.

8. DODATAK

Napomena: Zbog ograničenosti prostora pomoćne rutine (a i neki dijelovi programa) nisu priloženi nego su pod komentarom izložene njihove funkcije.

Algoritam 1

```
program algoritam_1 (input,output);
```

```
Label 999;
TYPE zadatak_status=(spreman_Z,aktivan_Z,kompletiran_Z);
     zadatak = zadatak_rec;
     zadatak_rec = RECORD
                    zadatak_ID: integer;
                    prioritet: integeri;
                    Zaht_mem: integer;
                    status: zadatak_status;
                    sljedeći:zadatak;
                   END;
     VP_status = (aktivan_VP,slobodan_VP);
     Virt_Proc = Virt_Proc_rec;
     Virt_Proc_rec = RECORD
                      Virt_Proc_ID: integer;
                      status: VP_status ;
                      sljedeći: Virt_Proc
                     END;
VAR Raspoloživa_mem: integer;
    lista_spremnih_zadataka,l_sp_Z: zadatak;
    lista_slobodnih_VP,l_sl_VP: Virt_Proc;
BEGIN
l_sp_Z :=lista_spremnih_zadataka;
l_sl_VP:=lista_slobodnih_VP;
     WHILE (l_sp_Z <> nil) AND (l_sl_VP <> nil) DO
      BEGIN
        IF l_sp_Z↑.status = spreman_Z THEN
        BEGIN
          IF l_sp_Z↑.Zaht_mem <= Raspoloživa_mem THEN
          BEGIN
            l_sl_VP :=l_sl_VP↑.sljedeći
            Raspoloživa_mem:=Raspoloživa_mem-l_sp_Z↑.Zaht.
            mem;
            l_sp_Z.status :=aktivan_Z;
            (* AKTIVIRAJ(l_sp_Z, l_sl_VP *)
          END
          ELSE
             GOTO 999
       END;
       l_sp_Z :=l_sp_Z↑.sljedeći;
999:END;
END.
```

Algoritam 2

```
program algoritam_2 (input,output);

TYPE zadatak_status=(spreman_Z,aktivan_Z,kompletiran_Z);
     zadatak = zadatak_rec;
     zadatak_rec = RECORD
                    zadatak_ID: integer;
                    prioritet: integer;
                    Zaht_mem: integer;
                    status: zadatak_status;
                    sljedeći: zadatak;
                   END;
     VP_status = (aktivan_VP,slobodan_VP);
     Virt_Proc =↑Virt_Proc_rec;
     Virt_Proc_rec = RECORD
                      Virt_Proc_ID: integer;
                      status: VP_status;
                      sljedeći: Virt_Proc
                     END;

VAR Raspoloživa_mem: integer;
    lista_spremnih_zadataka,l_sp_Z: zadatak;
    lista_slobodnih_VP,l_sl_VP: Virt_Proc;

BEGIN
l_sp_Z:=lista_spremnih_zadataka;
lista_slobodnih_VP:=l_sl_VP;
  WHILE(l_sp_Z <> nil) AND (l_sl_VP <> nil) DO
   BEGIN
     IF (l_sp_Z↑.status = spreman_Z) AND
        (l_sp_Z↑.Zaht_mem <= Raspoloživa mem)THEN
   BEGIN
     Raspoloživa mem:=Raspoloživa_mem-l_sp_Z↑.Zaht_mem
     l_sp_Z↑.status:=aktivan Z;
     (* AKTIVIRAJ (l_sp_Z,l_sl_VP *)
     l_sl_VP:=l_sl VP↑.sljedeći
   END;
   l_sp_Z:=l_sp_Z↑. sljedeći;
END;
```

Algoritam 3

```
program algoritam_3 (input,output);

TYPE zadatak status=(preemtivan_Z,spreman_Z,aktivan_Z,
                     aktiviranje_Z_u_toku, preem_Z_u_toku,
                     kompletiran_Z);
     zadatak = zadatak
     zadatak_rec = RECORD
                    zadatak_ID: integer;
                    prioritet: integer;
                    Zaht_mem,integer;
                    status: zadatak_status;
                    sljedeći: zadatak;
                   END;
VP_ status   = (preemptivan_VP, nepreemptivan_VP, aktivan
               _VP, slobodan_VP);
Virt_Proc =↑Virt_Proc_rec;
Virt_Proc_rec = RECORD
                 Virt_Proc_ID: integer;
                 status: VP_status;
                 aktivni_zadatak: zadatak_rec;
                 sljedeći: Virt_Proc
                END;
VAR Tren_Mem_raspol_za_raspor: integer; (* u nju ne ulazi
                                        memorija koju za-
                                        uzimaju preemptivni zadaci *)
    Sum_Mem_raspol_za_raspor: integer; (* u nju ulazi memorija koju zauzimaju preemptivni zadaci *)
    Tren_br_VP_raspol_za_raspor:integer; (* u njih ne ulaze VP aktivni na preemptivnim zadacima *)
    Sum_br_VP_raspol_za_raspor:integer; (* u njih ulaze VP aktivni na preemptivnim zadacima *)
    lista_aktivnih_VP: Virt_Proc; (* lista aktivnih VP-a na kojoj su VP poredani prema prioritetu aktivnih zadataka na njima i to od najnižeg do najvišeg nivoa prioriteta *)
    lista_spremnih_zadataka: zadatak; (* lista spremnih zadataka poredanih po prioritetu)*)
    Preem_mem, Preem_VP: integer;
    Mem_za_koju_je_preem_u_toku, Br_VP_za_koje_je_preem_u
    _toku: integer;
    Sum_zaht mem: integer;
    Zaht_br_VP: integer;
Sum_mem, Tren_mem, Sum_VP,Tren_VP: integer;
l_a_VP, l_sl_VP: Virt_proc;
l_sp_Z,l_a_Z: zadatak;
BEGIN
Tren_Mem_raspol_za_raspor :=100;
Tren_br_VP_raspol_za_raspor :=5;
l_a_VP :=lista_aktivnih_VP;
l_sl_VP:=lista_slobodnih_VP;
Sum_mem:=Sum_Mem_raspol_za_raspor;
Tren_mem:=Tren_Mem_raspol_za_raspor;
Sum_VP :=Sum_Br_VP_raspol_za_raspor;
Tren_VP :=Tren_Br_VP_raspol_za_raspor;
WHILE l_a_VP <> nil DO
BEGIN
 IF l_a_VP↑.status = nepreemptivan_VP THEN
 BEGIN
  Sum_Mem :=Sum_Mem-l_a_VP↑.aktivni_zadatak.Zaht_mem;
  Sum_VP :=Sum_VP-1;
 END
 ELSE
 BEGIN
   IF l_a_VP↑.status = preemptivan_VP THEN
      l_a_VP↑.aktivni_zadatak.status  :=preemtivan_Z
 END;
l_a_VP :=l_a_VP↑.sljedeći;
END;
(* pretraži listu spremnih zadataka *)
Mem_za_koju_je_preem_u_toku : = 0;
Br_VP_za_koje_je_preem_u_toku :=0;
Sum_zaht_mem :=0;
```

```
 1sp_Z:=lista_spremnih_zadataka;
WHILE (1_sp_Z<>nil) AND (Sum_VP > 0) DO
BEGIN
  IF 1_sp_Z↑.Zaht_mem <= Sum_Mem THEN
  BEGIN
    IF 1_sp_Z↑.status = preemtivan_Z THEN
    BEGIN
      Sum_VP :=Sum_VP-1 ;
      Sum_Mem :=Sum_Mem-1_sp_Z↑. Zaht_mem;
      1_sp_Z↑.status :=aktivan_Z;
    END
    ELSE
    (* ako je zadatak spreman *)
     IF 1_sp_Z↑.status = spreman_Z THEN
       BEGIN
       (* za pronalaženje broja elemenata u listi
        korišten funkcijski podprogram Broj_elem_u_li-
        sti (Lista) *)
        Tren_VP :=Broj_elem_u_listi (1_sl_VP)
        Tren_VP :=Tren_VP-1;
        1_sl_VP :=1_sl_VP↑.sljedeći;
        Sum_Mem : = Sum_Mem-1_sp_Z↑.Zaht_mem;
        (* ako su resursi raspoloživi *)
         IF (1_sp_Z↑.Zaht_mem <= Tren_Mem) AND
             (Tren_VP > 0) THEN
         BEGIN
          1_sl_VP :=1_sl_VP↑. sljedeći
          Tren_VP :=Tren_VP-1 ;
          Tren_Mem:=Tren_Mem-1_sp_Z↑. Zaht_mem;
          1_sp_Z↑.status :=aktiviranje_Z_u_toku;
          1_sl_VP :=lista_slobodnih_VP;
          1_sl_VP :=1_sl_VP↑.sljedeći;
          1_sl_VP↑.status :=nepreemptivan_VP;
          (* Aktiviraj zadatak sa čela 1_sp_Z na pro-
           cesoru sa čela 1_sl_VP pomoću pomoćne pro-
           cedure AKTIVIRAJ (1_sp_Z, 1_sl_VP) *)
          END
          ELSE
          (* ako su resursi nisu raspoloživi *)
          BEGIN
           Zaht_br_VP :=Zaht_br_VP + 1 ;
           Sum_zaht_mem :=Sum_zaht_mem + 1_sp_Z↑. Zaht
          _mem;
          END
        END
        ELSE
        (* ako je preemptiranje zadataka u toku *)
         IF 1_sp_Z↑.status = preem_Z_u_toku THEN
         BEGIN
          Br_VP_za_koje_je_preem_u_toku :=Br_VP_za
                                          _koje_je_preem
                                           _u_toku + 1;
          Mem_za_koju_je_preem_u_toku :=Mem_za_koju_je
                                         _preem_u_toku +
                                         1_sp_Z↑.Zaht_mem;

         END;
         1sp_Z :=1_sp_Z.
END
(* izvrši preemptiranje *)
Preem_mem : = Tren_Mem + Mem_za_koju_je_preem_u_toku;
Tren_VP :=Broj_elem_u_listi (1_sl_VP)
Preem_VP :=Tren_VP + Br_VP_za_koje_je_preem_u_toku;
WHILE (1_a_VP<>nil) AND ((Preem_mem<Sum_zaht_mem)OR
(Preem_VP < Zaht_br_VP)) DO
BEGIN
  IF 1_a_VP↑.aktivni_zadatak.status =preemtivan_Z THEN
  BEGIN
    Preem_VP :=Preem_VP + 1;
    Preem_mem :=Preem_mem + 1_a_VP↑.aktivni_zadatak
                .Zaht_mem;
    1_a_VP↑. aktivni_zadatak.status   :=preem_Z_u_toku;
    1_a_VP↑.status :=nepreemptivan_VP;
    (* Preemptiraj zadatak sa cela liste 1_a_VP pomoću
     pomoćne procedure PREEMPTIRAJ (1_a_VP↑.aktivni_
     zadatak, 1_a_VP  *)
  END
  1_a_VP :=1_a_VP↑.sljedeći;
END
end
END.
```


LITERATURA

1) B.W.Unger, A computer resource allocation model with some measurement and simulation results", IEEE Transactions on computers C-26 (3) 243-259 (1977).

2) Ana Hac. "The comparison of SIMULA 67 Pascal and Fortran languages on the example of operating system simulation", Informatyka, 2 (1979).

3) Ana Hac. "Computer System Simulation in Pascal", Software-Practice and Experience, 12 777-784 (1982).

4) M.Ruschitzka and R.Fabry, A unifying approach to scheduling", CACM, 20, No.7, 469-477 (1977).

5) J.Larmouth. Scheduling for share of the machine, Software - Practice and experience, 5, No.1, 29-49 (1975).

6) VAX 11/780 VMS Pascal reference manual, DIGITAL.

VEČPROCESORSKI SISTEMI

BRANKO MIHOVILOVIĆ,
PETER KOLBEZEN

UDK: 681.519.7

UNIVERZA EDVARDA KARDELJA,
LJUBLJANA
INSTITUT JOŽEF STEFAN,
LJUBLJANA

Multiprocesiranje in pomnilniki zelo velikih kapacitet predstavljajo osnovo superračunalnikov. Enostaven Ware-ov model večprocesorskih sistemov (VP) potrjuje dejstvo, da šele VP sistemi predstavljajo osnovne gradnike inteligentnih računalnikov. V nekaj odstavkih so na kratko naštete nekatere osnovne arhitekture današnjih in bodočih VP sistemov; podane pa so tudi osnovne karakteristike sistemske programske opreme večprocesorskih sistemov.

MULTIPROCESSOR SYSTEMS- Multiprocessing and memories with large capacitetes are are base of supercomputers. The simple Ware model of multiprocessor systems (VP) verifie the fact that MP systems represent the basic constructions of intelligent computers. In some sections some basic architectures of today and future MP systems are done. The basic characteristic of System software of MP systems are done too.

Uvod

Čeprav je zamisel o paralelnem procesiranju že zelo stara, lahko zanesljivo trdimo, da bo predstavljalo paralelno procesiranje osnovo naslednje generacije računalnikov. Nagel razvoj VLSI tehnologije ter dobro uporabljeno paralelno procesiranje bo omogočalo gradnjo (ali celo nadomeščanje) učinkovitih algoritmov, gradnjo zelo sposobne materialne in programske opreme VP sistemov.

Naravno nadaljevanje razvoja hitrega in visoko zmogljivega enoprocesorskega sistema predstavlja takoimenovan asinhron VP sistem, ki ga sestavlja nekaj šibko povezanih procesorjev. Takšni sistemi, z dvema do štirimi procesorji, so danes povsem komercialno dostopni (Cray-2). Obdobje do leta 1990 bo predstavljalo pomembno prelomnico v razvoju računalnikov. Večprocesorski sistemi s šestnajstimi procesorji bodo postali večprocesorski standard. Takšni sistemi bodo eni od osnovnih gradnikov bodočih VP sistemov, ki bodo v polni meri izkoriščali možnost sočasnega izvajanja opravil. Predstavljali bodo eno od arhitekturnih značilnosti računalnikov 5. generacije z nekaj tisoč procesorjev.

Ware-ov model VP sistema

Očitna razlika med enoprocesorskim in večprocesorskim sistemom je ta, da so slednji hitrejši. Koliko so eni sistemi hitrejši od drugih lahko ocenjujemo s faktorjem pohitritve, ki ga označimo z S. Faktor S definiramo z naslednjim razmerjem:

$$S = \frac{\text{čas izvajanja opravil na enem procesorju}}{\text{čas izvajanja opravil na N procesorjih}}$$

Oglejmo si, kakšen je faktor pohitritve VP sistema s šibko povezanimi procesorji na takoimenovanem Ware-ovem modelu.

Vzemimo, da je nek proces grajen tako, da ga lahko razdelimo v n opravil, katera bi enoprocesorski sistem izvedel v času nt. K opravil je takšnih, da so med seboj neodvisna in se lahko sočasno izvajajo na p procesorjih. 1-k opravil pa je takšnih, da so v medsebojni odvisnosti, zato se ne morejo izvajati sočasno (temveč le sekvenčno *. VP sistem izvede n opravil v času

$$(1-k)nt + knt/p \quad 0 < k \leq 1$$

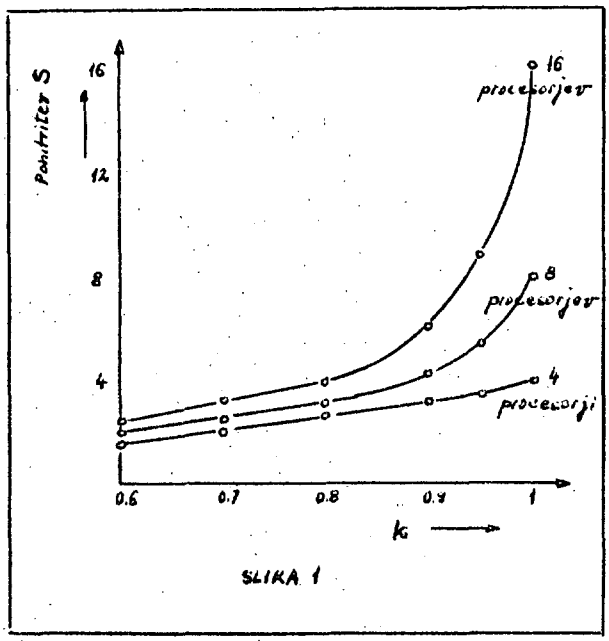
Dobljene rezultate primerjajmo glede na faktor pohitritve:

$$S(p,k) = \frac{nt}{(1-k)nt + knt/p} = \frac{p}{(1-k)p + k}$$

*Zavedati se moramo predpostavke, da v času t lahko deluje v VP sistemu en sam procesor, ali vseh p procesorjev v VP hkrati.

Zanima nas, kako se faktor S spreminja v odvisnosti od k, odnosno kakšen je faktor pohitritve pri popolni sočasnosti, ko je k=1. Parcialni odvod S po k nam da naslednji rezultat:

$$\frac{S(p,k)}{k}\Big/_{k=1} = p^2 - p$$



SLIKA 1

Slika 1 prikazuje Ware-ov model pohitritve kot funkcijo k-ja in sicer za večprocesorske sisteme z 4, 8 ali 16 procesorji. Na diagramu takoj opazimo, da je pohitritev razmeroma majhna za k, ki je manjši od 0.9. To pomeni, da k občutni pohitritvi pripomorejo le zelo učinkoviti paralelni algoritmi. Takšni algoritmi nikakor ne morejo biti načrtovani splošno (univerzalno), kajti veličina k je odvisna od notranje strukture VP sistema, ki pa je lahko povsem uporabniško zasnovana.

Opisani model je grajen na predpostavki, da je tok instrukcij, ki se izvaja na paralelnem sistemu, enak toku instrukcij, ki se izvaja na enoprocesorskem sistemu. Za sinhronizacijo odnosno komunikacijo med procesorji v VP sistemu skrbijo posebne instrukcije, ki jih v ta namen vgrajujemo v paralelne algoritme.

Ware-ov model nekoliko modificiramo tako, da času delovanja VP sistema prištejemo funkcijo δ(p). Funkcija mora biti monotono naraščajoča in nenegativna.

δ ni le funkcija p-ja, temveč je tudi funkcija strukture algoritma, arhitekture sistema in predvsem veličine k. Označimo s S(p,k,δ) pohitritev za preurejen Ware-ov model. Tako velja:

$$S(p,k,\delta) = \frac{1}{(1-k)+ k/p + \delta(p)}$$

Za popolno sočasje velja naslednji izraz:

$$S(p,k,\delta) = \frac{p}{k=1 \quad 1+ p\delta(p)}$$

ali z drugimi besedami, maksimalna pohitritev realnega VP sistema je vselej manjša kot je število procesorjev p. Na pohitritev S ne moremo uplivati z večanjem števila procesorjev. Razmerje k/p postane namreč zanemarljivo majhno. Vidimo, da je možno vplivati na pohitritev le z funkcijo δ(p).

Opisan model kaže, da je bil v preteklosti razvoj na področju paralelnega procesiranja usmerjen v iskanje učinkovitih algoritmov, programskih jezikov in paralelnih arhitektur. Nekaj let nazaj je bilo paralelno procesiranje predvsem domena znanstvenih ustanov. Ta čas eksperimentalni, namensko zgrajeni paralelni računalniki (reaktorska tehnika, različne simulacije v fiziki, reševanje kompleksnih matematičnih problemov, itd.) so bili skromni po številu procesorjev (majhen k), zato pa so bili namensko razviti tako programska oprema kot ustrezni algoritmi (velik δ) izredno bogati. Vsekakor je željam po vse večji hitrosti procesiranja in učinkoviti grafični, odnosno tabelarični predstavitvi podatkov mogoče zadostiti le z VP sistemi z zadostnim številom procesorjev.

Stanje in perspektive VP sistemov

Profesor P. C. Treleaven je na enem od simpozijev o računalniških arhitekturah napovedal dve osnovni skupini bodočih računalniških arhitektur: V prvo sodijo takoimenovani VLSI VP sistemi, ki v bistvu predstavljajo povezovanje danes že standardnih mikroprocesorjev s potrebnimi VLSI elementi v mikroračunalniške module tako, da bodo le-ti, kot tudi pomnilniški elementi že vnaprej pripravljeni za povezovanje v večprocesorske sisteme.

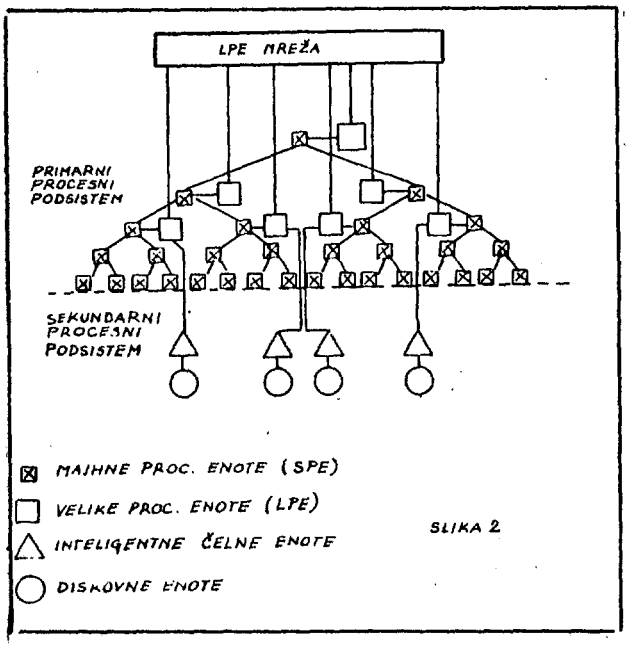
V drugo skupino sodijo integrirane mreže računalnikov, ki pomenijo združitev večjega števila računalnikov v enote ali celice, ki pa bodo standardizirane glede na razdalje med računalniki v celici. Takšne integrirane mreže računalnikov omogočajo učinkovitejše programiranje, izvajanje programov in komuniciranje z drugimi enotami.

Isti profesor je navedel še dva osnovna mehanizma, katerima bodo podrejene vse bodoče arhitekture visoko sposobnih računalnikov. To sta podatkovni in upravljalski mehanizem. Podatkovni mehanizem določa kako je nek argument udeležen v množici instrukcij. Upravljalski mehanizem pa določa, kako neka instrukcija pogojuje uzvajanje druge instrukcije.

Primer arhitekture, v kateri sta združena oba mehanizma je MIMD (multiple-instruction multiple-data-stream) arhitektura. MIMD računalnik sestavlja večje število neodvisnih procesorjev, ki so sočasno vodeni z različnimi instrukcijami nad različnimi tokovi podatkov.

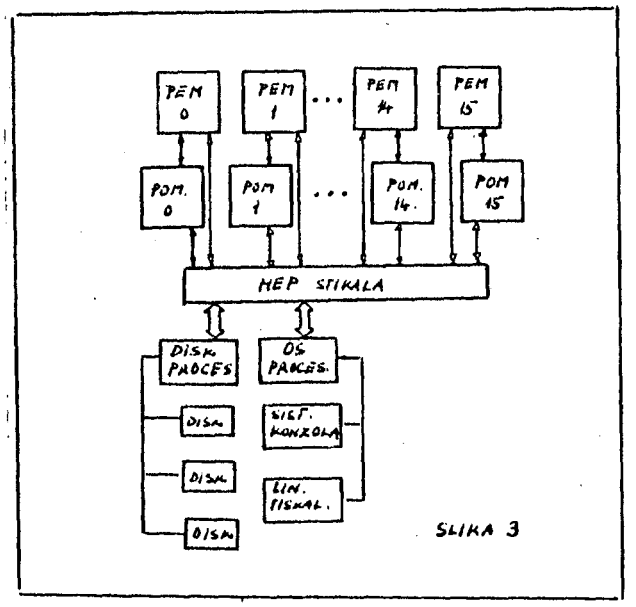
Na kolumbijski univerzi že nekaj let potekajo raziskave na družini paralelnih računalnikov tipa NON-VON, ki predstavljajo podrazred MIMD računalnikov. Karakteristika teh računalnikov je mešana struktura procesnih in pomnilniških skupin na različnih nivojih. Primarni elementi v tej strukturi so majhni, a učinkoviti procesni elementi, ki vsebujejo tudi do 64k

lastnega pomnilnika. Sekundarni procesni elementi v tej strukturi pa so veliki procesni podsistemi. Osnovo le-teh predstavljajo banke inteligentnih diskovnih pomnilnikov, ki so z primarnimi procesnimi sistemi povezani preko hitrih paralelnih prenosnikov.Osnovno skico NON-VON sistema prikazuje slika 2.



SLIKA 2

Opišimo še en primer MIMD paralelnega računalnika z imenom HEP procesor (Heterogeneous Element Procesor), ki je danes že komercialen proizvod. Karakteristika tega računalnika (shematično ga prikazuje slika 3) je sposobnost izvajanja 160x10 instrukcij/sekundo. Zgradba računalnika vsebuje večje število procesnih modulov (PEM). Vsak modul ima lastno pomnilniško banko in povezavo z operacijskim sistemom (ki ga poganja poseben procesor), in diskovnim sistemom preko stikalnega polja. Paralelno procesiranje se izvaja znotraj vsake PEM enote z pipeline instrukcijami tako, da se v vsakem



SLIKA 3

trenutku izvajajo instrukcije na vseh enotah v različnih fazah. Če npr. ena instrukcija zahteva 8 ciklov (pomnilniška 12), pomeni, da lahko sočasno teče 8 (12) procesov. Problem konfliktnih situacij med PEM enotami je rešen z medsebojno povezavo lastnih pomnilniških bank preko posebnih hitrih stikal. Čas reševanja neke naloge na MIMD računalniku z p procesorji (pod pogojem, da je p≤12) je kar obratno proporcionalen številu uporabljenih procesorjev.

Funkcionalni jeziki

Pri aplikativnem programiranju večprocesorskih sistemov, igrajo pomembno vlogo funkcionalni jeziki, ki omogočajo izkoriščanje implicitnih sočasnosti v teh jezikih. Funkcionalni jeziki so neodvisni od hitrosti računalnika, števila uporabljenih procesorjev med izvajanjem programa, kot tudi od medsebojnih komunikacij med njimi. Aplikativno programiranje VP sistemov je zahtevno delo, saj se od programerja dodatno zahteva iskanje sočasnosti v programih in povezav le-teh s strukturo računalnika. Precej nesmoterno bi bilo npr. sekvenčno izvajanje skupine opravil na VP sistemu, če obstaja možnost, da se ta opravila izvajajo paralelno. Funkcionalni jeziki so zagotovo idealni za programiranje večprocesorskih sistemov saj tvorijo najbližji prehod od programiranja enoprocesorskih sistemov na programiranje VP sistemov.

Klasično povezavo procesor-pomnilnik imenujemo tudi von Neumannovo ozko grlo. Več takšnih "ozkih grl" lahko povežemo v sistem, ki pa je lahko z uporabo funkcionalnega jezika učinkovitejši od kateregakoli VP sistema drugačne konfiguracije. Večja učinkovitost sistema je pogojena tudi z takšno organizacijo, ki omogoča pogostejše komunikacije med procesnimi pari (procesor/pomnilnik), kot pa med posameznimi komponentami sistema.

Inherentne paralelnosti v funkcionalnih jezikih torej v veliki meri odražajo arhitekturo VP računalnika (s tem je ta prezentirana programerju), ki pri programiranju le-to tudi upošteva. Paralelni računalnik je lahko grajen tako, da na učinkovit način "spremlja" eno od karakteristik funkcionalnih jezikov in to je lokalnost učinka. Slednje pomeni, da se med procesiranjem podatki ne spreminjajo v tistem delu pomnilnika, ki je tem podatkom namenjen, temveč se podatki spreminjajo (oblikujejo) sprotno v točkah procesiranja.

Operacijski sistemi VP sistemov

Poleg aplikativnega dela sistemske programske opreme imamo, tako kot v enoprocesorskih sistemih, še drug del in to je operacijski sistem.

Veliko rezultatov raziskav multiprogramskih sistemov je direktno uporabljivih na VP sistemih; še več, naloga celotne programske podpore VP sistema je v tem, da omogoča, ob izvajanju programa, avtomatično eksploatacijo paralelizma v materialni opremi. Uspešno ...procesiranje je odvisno ne le od parametrov, ki so domena multiprogramskih sistemov (porazdelitev procesov, medprocesne

komunikacije), temveč tudi od nadzora nad povezavami med komponentami sistema, kar pa je domena operacijskega sistema (OS).

V konceptu obstajajo majhne razlike med sistemsko programsko opremo VP sistema in sistema, ki uporablja multiprogramiranje. Operacijska sistema obeh sistemov vsebujeta naslednje:

-Razporeditev virov in upravljanje
-Varovanje podatkovne osnove
-Preprečevanje sistemskih deadlock-ov
-Prekinjevanje izvajanja nenormalnih procesov
-Prerazporejanje vhodno/izhodnih bremen
-Prerazporejanje obremenjevanja procesorjev
-Rekonfiguracija sistema.

Zadnje tri navedbe so specifične, njihova izvedba pa je zelo pomembna za OS VP sistemov, kajti preko njih se zrcali zahtevana velika učinkovitost OS VP sistemov.

Razlikujemo vglavnem tri organizacije OS VP sistemov. To so: master-slave, ločeno izvrševanje za vsak procesor,simetrična porazdelitev med procesorje.

Najbolj preprosta organizacija OS je vsekakor prva, saj jo lahko od vseh treh vrst OS najenostavneje realiziramo. V bistvu gre za razširitev OS enoprocesorskega sistema na ta način, da je v polni meri uporabljeno multiprogramiranje. Karakteristike tega OS so naslednje: Programe OS izvršuje vselej samo eden za to namenjen procesor, ki krmili ostale procesorje v sistemu. Na ta način je delno rešen problem konfliktnih situacij v sistemu, vendar izpad master enot pomeni popoln izpad celotnega sistema. Sistem je dokaj nefleksibilen, v slave delu sistema pa lahko pride do precejšne izgube časa in čakalnih vrst. Sicer pa je lahko tako materialna kot programska oprema dokaj preprosta, sistem pa je lahko zelo učinkovit v tistih aplikacijah, kjer je delo slave procesorjev popolnoma definirano.

Ime za drugo vrsto OS je nekoliko nerodno, pomeni pa, da imajo vsi procesorji enake lastne OS. Vsak procesor ima lahko svoje vhodno/izhodne elemente, masovne pomnilnike, podatkovno osnovo ali pa je slednja vsem skupna. Pri vsem tem pa obstaja možnost pojava konfliktnih situacij v sistemu. Sistem ni občutljiv na katastrofalne napake, vendar zaradi v/i elementov ni primeren za morebitno rekonfiguracijo.

Tretja vrsta OS je do neke mere izpeljanka predhodne, predstavlja pa za realizacijo najtežavnejšo vrsto. Pri tej vrsti govorimo o takoimenovanem plavajočem nadzoru, kar pomeni, da lahko katerikoli procesor prevzame vlogo nadzornega modula v sistemu. Tako je lahko zagotovljena enakomernejša obremenitev virov sistema. Konfliktne situacije se rešujejo z metodo prioritet, ki je v sistem vnešena statično, ali pa se le-ta dinamično spreminja. No, prednost te vrste OS je možnost reduciranja sistema do uporabne velikosti vedno kadar s tem zagotovimo boljšo izkoriščenost reduciranega sistema, odnosno posameznih virov.

Namesto zaključka

Na koncu opozorimo še na dva ključna problema, ki se pojavita pri medsebojnem povezovanju večjega števila procesorjev. Prvič, s tem, da smo pridobili na "procesni moči" sistema, se zastavlja vprašanje, kako uporabiti VP sistem pri reševanju "enojnih"problemov; in drugič, kako zagotoviti ustrezne komunikacije med procesorji in pomnilniki.

Rešitev prvega problema zahteva prestrukturiranje programov ali algoritmov, kar naj bi v idealnem primeru potekalo avtomatsko. Drug problem naj bi navidezno rešili že s samo zgradbo VP sistema, vendar to ne zadostuje. Komunikacije med samimi procesorji ter procesorji in pomnilniki morajo biti tako grajene, da se učinkovito odražajo na delo, ki ga VP sistem opravlja. To pomeni, da tako načrtovalec materialne kot načrtovalec sistemske programske opreme ne moreta reševati komunikacijskih problemov ločeno, temveč je potrebno kompleksen problem komunikacij reševati na tak način, da se zagotovi čim večja učinkovitost VP sistema.

Literatura

[1] P. H. Enslow jr.
Multiprocessor Organization-A Survey
Computing Surveys, Vol.9, No.1,
Marec 1977

[2] S. I. Kartashev, S. P. Kartashev
Dynamic Architectures: Problems and Solutions.
Computer, Julij 1978

[3] D. Highberger
Intelligent computing era takes off
Computer Design, September 1984.

[4] J. Šilc, B. Robič
Osnovna načela DF sistemov
Informatica 1, Januar 1985.

[5] N. Mokhoff
Parallelism makes strong bid for next generation computers
Computer Design, September 1984.

[6] B. Buzbee
Parallel processing makes tough demands
Computer Design, September 1984.

[7] B. Mihovilovič, P. Kolbezen
Paralelno izvajanje opravil v VP sistemu, 1. del
Informatica 3, 1984.

[8] B. Mihovilovič, P. Kolbezen
Paralelno izvajanje opravil v VP sistemu, 2. del
Informatica 3, 1984.

INFORMATICA 2/85

KLASIČNI ALGORITMI ZA ISKANJE MINIMALNIH DREVES

KRSTE JOVANOSKI

UDK: 519.698

ELEKTROINŠTITUT „MILAN VIDMAR“, LJUBLJANA

V članku je obravnavan problem iskanja minimalnih vpetih dreves. Podana sta dva klasična algoritma (Primov in Kruskalov) za iskanje minimalnih vpetih dreves povezanega grafa. Poleg tega je podana analiza časovne kompleksnosti omenjenih algoritmov. Navajamo najbolj preproste algoritme, ki rešujejo zgornji problem. Podana je tudi ocena učinkovitosti algoritma v odvisnosti od (števila točk in števila povezav) grafa, ter praktični nasveti za njuno uporabo. Med drugim je opisana še strategija za načrtovanje algoritmov znana pod imenom požrešna metoda.

CLASSICAL ALGORITHMS FOR FINDING MINIMUM SPANNING TREES. The paper is concerned with the problem of finding the minimum spanning tree in the connected graph. Two algorithms are given (the Prim's and the Kruskal's) for finding minimum spanning trees. Analysis of the time complexity of the discussed algorithms is also given. The simples algorithms, applicable for solving the problem, are outlined. Estimation of efficiency of the procedure as a function of the graph (number of vertices and number of edges) along with practical advices regarding their application is also presented. The author tackles also the question of strategy known as 'greedy method'.

Kot zgled za uporabo minimalnega vpetega dreves a si mislimo reševanje problemov povezanih električnih (računalniških, telefonskih, daljnovodnih, televizijskih), cestnih, železniških, vodovodnih, plinskih ali toplovodnih omrežjih. Rešitev zgornjega problema, nam lahko da najbolj varčno investicijo. Začnimo kar z definicijo problema.

Def.: Naj bo G=(V,P) povezan graf.

$$c : P \longrightarrow R^{+}$$

Naj bo T povezan vpet podgraf grafa G, ki ima minimalno vsoto

$$\min\left(\sum_{e \in E(T)} c(e)\right)$$

potem pravimo T-ju minimalno vpeto drevo.
Zgornji problem bomo reševali s požrešno metodo . Požrešna metoda je preprosta strategija, ki običajno vodi v učinkovit postopek.

Imejmo množico z n elementi ali kazalci nanje,

$$\{ a[i] ; i \in J \} , \quad J \subseteq \{1,2,3,\ldots,n\} ,$$

ki zadošča danim zahtevam in optimizira kriterijsko funkcijo f. Podmnožici, ki zadošča zahtevam, rečemo dopustna. Dopustna množica, ki optimizira kriterijsko funkcijo, je optimalna. Osnovna misel strategije je: Rešitev gradimo postopoma. Na tekočem koraku poiščemo element, za katerega bi bila najbolj navdušena kriterijska funkcija. Sprejmemo ga le, če je tudi s tem elementom razširjena množica dopustna. Problem minimalnega vpetega drevesa bomo reševali, tako da bomo izbirali postopoma povezavo za povezavo. Naj bo D podmnožica povezav omrežja G, ki smo jo zgradili do danega koraka. Naj bo D' množica povezav, tako da je

$$D' = D \cup \{p\} \quad p := \text{nova privzeta povezava}$$

bo dopustna , če v D' ni cikla. Za vrstni red izbiranja uporabimo vrednost povezave. Postopek, ki gradi rešitev, tako, da je rešitev vseskozi

drevo imenJuJemo Primov postopek. Postopek, ki gradi rešitev, tako, da je rešitev vseskozi goz d, ki le na koncu preraste v drevo imenJuJemo Kruskalov.

1. Primov algoritem

Rešitev gradimo kot drevo. Tekoči kandidat je p ovezava, ki veže vozlišče drevesa z vozliščem, ki še ni v drevesu in ima minimalno vrednost. V ozlišča grafa G na tekočem koraku razpadejo na dve disJunktni množici, na vozlišča tekočega dr evesa D in preostanek. Povezava, ki jo privzame mo, ima eno vozlišče v drevesu, drugo pa v preo stanku grafa. Za dano vozlišče v∈E(D) pride v t ekočem koraku v izbor najkrajša od povezav, ki se začno v vozlišču v in končajo v kateremkoli vozlišču tekočega drevesa.
ToreJ je povsem dovolJ, da poznamo k vozlišču v najbližJe vozlišče v tekočem drevesu D, če le l ahko od tod določimo dolžino povezave v konstan tnem času. Potrebujemo naslednJe operacije

1) Ali je vozlišče že v D ?
2) Katero vozlišče drevesa D je k v najbližJe ?
3) Popravi tekočo rešitev D.

Graf bomo predstavili z matriko c.

c[i,J]= vrednost povezave z krajišči v in v .
i J

Vektor r bo vseboval indeks najbližJega soseda.

r: V -> V

$$J \mapsto r[J] := \begin{cases} 0 & J \in V \\ t & \min_{i \in V} \{ c[J,i] \} = c[J,t] \end{cases}$$

Časovna zahtevnost postopka je $O(n^2)$ saJ sta dve n korakov dolgi zanki vgnJeni druga v drug o.

2. Kruskalov algoritem

Razvijemo še drugi postopek. Po privzetku gradi mo gozd in tekočo najceneJšo povezavo privzame mo, če ne naredi zanke. To pa bo natanko tedaJ, ko obe vozlišči povezave (u,v) nista v istem d revesu, v isti komponenti.
Operaciji, ki Ju potrebujemo sta

1) Ugotovi v kateri komponenti grafa je dano vo zlišče.
2) Spoji dve komponenti v novo.

Preden izpeljemo postopek, si še izberimo preds tavitev grafa. Postopek zahteva, da povezave ur edimo po nepadaJočih vrednostih. Povezave grafa in njihove vrednosti našteJemo v seznamu o s komponentami.

```
procedure minimalno vpeto drevo po Primu ( n,c,vrednost,drevo );
{ procedura poišče v grafu z n vozlišči in z vrednostmi povezav
                c[i,J] >= 0,   i,J = 1,2,...,n - 1
  in vrednostJo vrednost. Če drevesa ni mogoče poiskati, je vrednost
  neskončno.}
Begin
      (k,l) := povezava z minimalno vrednostJo ;
      vrednost := c[i,k];  { drevo[1,1],drevo[1,2] := (k,l)};
      for i := 1 to n do
          Begin
              poišči J ; 1 <= J <=n ,tako da je v[J] ≠ 0 in
                                        c[J,r[J]] minimalna  ;
              { drevo[i,1] , drevo[i,2] := (J,r[J])
              vrednost := vrednost + c[J,r[J]]) ;
              r[J] := 0 ;
              for k := 1 to n do
                 if r[k] ≠ 0 and c[k,r[k]] > c[k,J] then r[k]:=J
          End
End.
```

(i,J,c[i,J]) [1] ; 1 = 1,2,...,m,
kJer Je m = |P|.

Ker na tekočem koraku iz seznama brišemo vedno najkrajšo povezavo, sa organiziramo kot vrsto s prednostJo.
OsleJamo si časovno zahtevnost (kompleksnost) algoritma. Algoritem zahteva, da vnaprej vse ob stoJeče povezave razvrstimo v vrsto s prednostJo sleda na vrednosti povezav. Pri izbiri smisel ne predstavitve (kopica , 2-3 drevo, najbolJ le vo drevo, itd.) bo ta zahtevnost O(m * log(m)), m=|P|. Komponente tekoče rešitve vodimo kot mno žice točk. Če si izberemo učinkovito predstavit ev za množice, bosta operaciJi poišči in uniJa praktično tekli v konstantnem času, torej k cel oti prinesli kvečemu O(m).

Izrek : Kruskalov postopek Je časovne zahtevno
------ sti O(m * log(m)) .

Za graf, ki Je poln, Je

$$O(m * \log(m)) = O(n^2 \log(n))$$

in Primov postopek manJšega reda, za graf, v ka terem število povezav ni mnogo večJe kot število vozlišč, Je

$$O(m * \log(m)) = O(n * \log(n))$$

in razviti postopek usodnejši.

Izrek : PrejšnJi algoritem poišče minimalno
------- vpeto drevo.

Dokaz : NaJ bo G povezano, neusmerJen graf, D
------- množica povezav grafa, ki ga konstruira postopek, D' ≠ D pa množica povezav minimalnega vpetega drevesa.
Pokazati moramo, da velJa

vrednost(D) ≤ vrednost(D').

Oba podgrafa sta vpeta, zato imata enako število povezav,

|D| = |D'| = n - 1,

torej lahko poiščemo povezavo p, ki ima med vse mi povezavami aeD in aéD' minimalno ceno. Vstav

```
procedure minimalno vpeto drevo po Kruskalu ( n, m, vrednost, drevo);
{ Procedura sestavi minimalno vpeto drevo po Kuskalu. Graf Je podan kot vrst
  a s prednostJo m, v kateri so našteti podatki (i,J,c[i,J]) in ima najkr
  aJša povezava prednost. Končno drevo sestavlJa seznam začetnih in končni
  h vozlišč veJ drevesa
                ( drevo[i,1], drevo[i,2],  i = 1,2,...,n - 1,
 in Je vrednost vrednosti.
 Če drevesa ni mogoče sestaviti Je vrednost := neskončno }
Begin
    for i := 1 to n do oče[i] := -1 ;
    { vozlišča so na začetku vsako v svoJi komponenti }
    k := 0 ; vrednost := 0 ;
    While k < n - 1 and not prazna(m) do
       Begin
            briši((u,v,c[u,v]),m);
            { to Je povezava z minimalno vrednostJo }
            poišči (oče , up , vp);
            if up ≠ vp then
               Begin
                    { Nova povezava (u,v) ne naredi cikla }
                    k := k + 1 ;
                    (drevo[k,1],drevo[k,2]) := (u,v);
                    vrednost := vrednost + c[u,v];
                    { Brojimo komponenti, ki Ju (u,v) povezuJe. }
                    uniJa( oče , up , vp )
               End
       End ;
    if k > n-1 then vrednost := neskončno
End.
```

imo p v D'. V D' dobimo natanko en cikel, podan o na primer z zaporedjem povezav

p , p[1],...,p[r].

V tem zaporedju je vsaj ena povezava, recimo p[j], ki ni v D, saj bi drugače D vseboval cikel.
Trdimo , da je

c[p[j]] ≥ c[p].

Res, po izbiri p (p je najkrajša povezava!) s o vse povezave z manjšo ceno.

c[a] < c[p] , a∈D'

v D natanko tedaj kot v D'. Toda vse povezave a

c[a]<c[p], a∈D'

ne tvorijo cikla s p[j] in p[j], ki bi prišla p rej na vrsto kot p, če bi veljalo

c[p[j]] < c[p].

Sestavimo novo drevo D*,

D*:=D' U {p} - {p[j]},

za katerega po konstrukciji velja

vrednost(D*) ≤ vrednost(D').

Drevo D* se v eni povezavi več ujema z D. Postopek ponavljamo toliko časa, da optimalno drevo prevedemo na drevo, ki nam da Kruskalov postopek, ne da bi pri tem povečali njegovo vrednost. Dokaz je končan. ∎

Z A K L J U Č E K

Poleg klasičnih postopkov za reševanje problema o minimalnem vpetih drevesi, obstajajo še bolj učinkoviti postopki z ugodnejšo časovno zahtevnostjo. Uporabljajo bolj komplicirane strukture podatkov in njihova analiza je bolj zapletena. Zaradi tega sem se odločil predstaviti samo klasične algoritme. V zadnjih letih srečamo še paralelne algoritme. Uporaba se bo uveljavila z uporabo močnih večprocesorskih sistemov.

Z A H V A L A

Na koncu se zahvaljujem vsem, ki so mi pomagali in me spodbujali pri realizaciji tega članka. Posebej se zahvaljujem Dr. R.E.Tarjanu za koristne sugestije, Dr. J. Kozaku, Dr. T. Pisanskemu in V. Batagelju.

L I T E R A T U R A

1. J.B. Kruskal, Jr., On the shortest spanning subtree of a graph and the traveling salesman problem,
   Proc. Amer. Math. Soc., 7(1956), pp. 48-50
2. R.C.Prim, Shortest connection networks and some generalizations,
   Bell System Tech.J.,36(1957), pp. 1389-1401
3. R.E.Tarjan (Privatno sodelovanje).

INFORMATICA 2/85

GOVOR V KOMUNIKACIJI MED STROJEM IN ČLOVEKOM

DAVOR MILJAN IN JURIJ ŠILC

UDK: 681.3:007

INSTITUT „JOŽEF STEFAN"
LJUBLJANA

Podan je zgodovinski razvoj in današnje stanje na področju analognih in digitalnih govornih sistemov. Narejen je pregled nekaterih najbolj pogosto uporabljenih metod za sintezo govore iz skupine kodirnikov/dekodirnikov signalne krivulje (PCM, DPCM, ADPCM, DM in CVSD metoda) in iz skupine vokoderjev kot so LPC, formantna in kanalna metoda.

Machine - Man Comunication by Voice Historical evolution and state of art of analog and digital speech systems is given. We have rewiewed some most used methods for speech synthesis from grup of waveform coders (PCM, ADPCM DM and CVSD method) and vocoders (LPC, formant and chanel method).

Uvod

Govor je najnaravneši način človekovega komuniciranja, zato je razumljiva njegova starodavna želja, da bi strojem, ki jih je zgradil, dodal sposobnost govorne komunikacije. V našem stoletju, dobi telekomunikacij, je ideja o govorni komunikaciji med človekom in strojem pridobila na pomembnosti. V industrijsko razvitem svetu se je začelo obširno teoretično in praktično delo na področju govornih komunikacij, ki je kmalu rodilo prve sadove. Nerazvita tehnologija je bila bistvena omejitev pri doseganju boljših praktičnih rezultatov. Po letu 1978, ko je firma Texas Instruments dala na tržišče prvo ceneno LSI-vezje "Speak & Spell", se je začelo novo obdobje v razvoju govornih komunikacij stroj človek. V naslednjih letih se pojavljajo številni večji in manjši proizvajalci, ki ponujajo nova LSI-vezja, komplete na tiskanih ploščah, programske pakete za različne računalnike in sisteme za generiranje in razpoznavanje govora. Odkrivajo se nova in nova področja v profesionalni in v osebni uporabi govorne komunikacije s strojem, zlasti po razširitvi uporabe mikro in osebnih računalnikov.

Govorno komunikacijo med človekom in strojem delimo na tri osnovna področja: digitalne govorne sisteme, sisteme za razpoznavanje govora in sisteme za razpoznavanje govorca. V tem prispevku se bomo omejili na govorno komunikacijo v smeri stroj človek.

Nekatere osnovne prednosti in pomanjkljivosti govorne komunikacije stroj človek so razvidne iz tabele 1 [1].

Prednosti
- omogočena komunikacija po telefonu
- uporabnikov pogled ni zaseden
- zelo učinkovito pri informacijah o alarmih ali obvestilih
Pomanjkljivosti
- govor lahko moti uporabnikovo okolico
- učinki se zmanjšajo v hrupni okolici

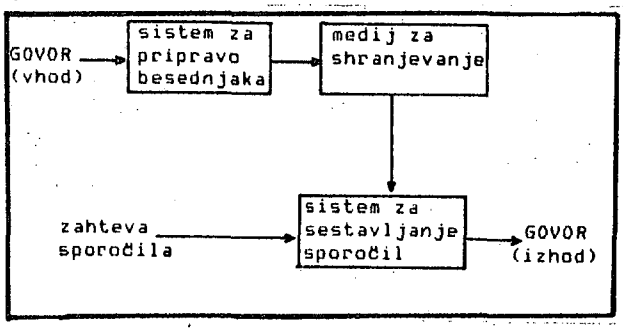
Tabela 1.

Analogni govorni sistemi

Tisočletja staro željo človeka, da bi izdelal govoreči neživi objekt, so poizkušali uresničiti že stari Grki in Rimljani [2]. Najprej, seveda v verske in mistične namene. Uporabljali so preprosto tehniko in tehnologijo; skrito napeljane cevke in parabolično grajene oboke, ki so prevajali, ojačevali in usmerjali glas skrbno skritega smrtnika.

Razvoj tehnologije je prinesel nove možnosti. Prvi začetki govornih avtomatov segajo v trideseta leta našega stoletja. Najprej so bili to analogni sistemi, pri katerih je besednjak predstavljala zbirka analogno posnetih besed naravnega govora [3] (Slika 1). Medij, na katerem je shranjen besednjak, je bil fotografski ali magnetni film, ki je

bil v obliki traku pritrjen na vrtečem se valju. Takšna, bolj ali manj izboljšana je večina še danes delujočih analognih govornih avtomatov. Od leta 1930 so se govorni avtomati uporabljali v telefoniji za posredovanje podatkov o točnem času, vremenu, pri klicu izklopljene telefonske številke in podobno ter pri različnih javnih avtomatih.



Slika 1: Analogni govorni sistem.

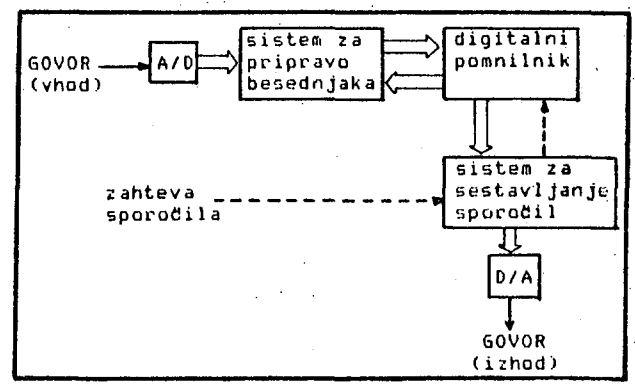
Izboljšave govornega avtomata so potekale v smeri hitrejšega dostopa do posameznih besed ter optimalnega števila shranjenih besed za čim večje število sestavljenih fraz. Hitrost dostopa se povečuje z večjim številom paralelnih trakov s čitalnimi glavami. Trakovi so razdeljeni na polsekundne segmente tako, da so shranjene besede časovno omejene, daljše besede pa se razdelijo na nekaj segmentov. Očitno je, da se z manjšo vrtilno hitrostjo valja povečuje količina zapisanih besed. Ker se valj vrti relativno počasi, morajo biti nekatere besede zaradi hitrejšega dostopa večkrat zapisane. Opisane izboljšave zahtevajo dodatno natančno krmiljenje večjega števila čitalnih glav po določenem vrstnem redu in časovni shemi. To nalogo opravljajo ustrezni avtomati ali v novejšem času računalniki.

Opisani analogni govorni avtomati se še vedno uporabljajo v primerih, ko besednjak sestavlja manjše število besed in ko se fraze sestavljajo po dovolj enostavnih shemah. Takšni sistemi imajo veliko omejitev. Omenimo naj premajhno prilagodljivost sistema, veliko natančnost pri izdelavi in obratovanju mehanskega sistema za reprodukcijo zvoka ter dolgotrajno spreminjanje besednjaka. Še opaznejša je slišna kvaliteta reproduciranih stavkov in fraz, ki jo pri takšnih sistemih le s težavo dosežemo. Fraze sestavljajo poljubne, ločeno posnete besede, zaradi česar nastajajo razlike v nivojih med koncem ene in začetkom naslednje reproducirane besede. Pojavljajo se problemi dinamike in naglas v fraze in podobni pojavi, ki negativno vplivajo na kvaliteto reprodukcije. *Takšne napake je mogoče* odpraviti z dolgotrajnim, pazljivim snemanjem besednjaka, za kar je potrebna ustrezna oprema. Opisane pomanjkljivosti in omejitve, ki tem sistemom ne omogočajo razvoja v smeri prilagodljivega, bogatega in naravno zvenečega govora, so botrovale razvoju digitalnih govornih sistemov.

Digitalni govorni sistemi

Postopek, po katerem so zgrajeni digitalni govorni sistemi prikazuje Slika 2. Očitna je velika podobnost z analognim sistemom, ki ga prikazuje Slika 1. Vhodna in izhodna stopnja digitalnega sistema sta A/D in D/A pretvornika, osrednji del pa je popolnoma digitalen. Digitaliziran naravni govor (digitalno zapisan besednjak) se hrani v digitalnem pomnilniku. Naloga sistema za sestavljanje sporočil je izbiranje digitalno zapisanih enot govora (glas, zlog, beseda, fraza) iz pomnilnika in generiranje poljubnih novih sestavljenih enot (beseda, fraza, stavek).

Postopek digitalnega shranjevanja in generiranja govora, kot je prikazan v splošnem diagramu, dopušča zelo različne postopke načrtovanja digitalnega govornega sistema. Ključni faktor je v načinu digitalnega shranjevanja govornih enot, ki sestavljajo besednjak. Izbira kodirnega postopka, zelo upliva na kvaliteto generiranega govora in velikost digitalnega pomnilnika, ki je potreben za shranjevanje besednjaka.



Slika 2: Digitalni govorni sistem.

Obstajata dva osnovna načina digitalne predstavitve naravnega govora. Prvič, neposredno kodiranje govorne valne oblike in drugič, analiza govorne krivulje, s katero se dobijo parametri, ki opisujejo analiziran govor in so v splošnem parametri za krmiljenje modela govornega trakta.

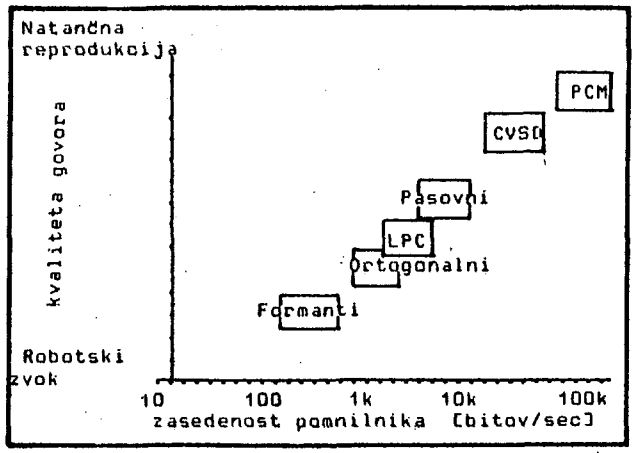
Objektivni kriteriji za vrednotenje digitalnih govornih sistemov še niso institucionalno standardizirani in rabijo le kot smernice načrtovalcem takšnih sistemov. V [32] so podani nekateri merljivi parametri, ki temeljijo na različnih S/N (signal to noise) razmerjih. Ocenjevanje kvalitete govornih naprav je v veliki meri subjektivno in se izvaja na številnih poskusih v obliki poslušanja strojno generiranega govora. Pri tem so v pomoč množice "standardnih" stavkov, ki so tako izbrani, da vsebujejo vse foneme nekega jezika v raznih kombinacijah. Pomembni so trije kriteriji: razumljivost in kvaliteta produciranega govora ter cena sistema.

Elementarni pogoj je razumljivost, ki pa mora biti pri velikem številu aplikacij več kot samo razumljivost. Kvaliteta govora ni v neposredni zvezi z razumljivostjo, je zgolj subjektivna ocena, ki pravi, da je visoko kvaliteten govor najbolj podoben naravnemu govoru, nekvaliteten pa je metalni - robotski govor. V [27] so definirane štiri stopnje kvalitete govornih sistemov:

- radiodifuzna, pri kateri dosežemo frekvenčno območje med 0 Hz in 7 KHz ob minimalnih popačitvah;
- telefonska, v katero sodijo sistemi, ki imajo frekvenčno območje med 200 - 3200 Hz, SNR ≥ 30 dB ter popačitev signala ≤ 2-3 % ;
- komunikacijska, ki še ohranja visoko razumljivost ob občutno slabši kvaliteti in večjih popačitvah;

- sintetična, ki jo dobimo pri t.i. vokoderjih ko govorni signal izgublja naravni zvok in deluje metalno-robotsko.

Ko doseže sistem zadovoljivo stopnjo razumljivosti, preostane načrtovalcem še izbira kompromisne rešitve glede cene in kvalitete govora, ob upoštevanju tipa aplikacije. Tako je na primer, pri bančnih avtomatih zaželjen zelo kvaliteten, naravnemu govoru podoben zvok, za razliko od govornih sistemov pri video in igralnih avtomatih, ko metalni zvok ne moti, ali je celo zaželjen.



Slika 3: Kvaliteta govora v odvisnosti od pretoka informacije.

V splošnem lahko rečemo, da je cena govornega sistema sorazmerna ceni pomnilnika. Različni postopki kodiranja ali parametriziranja govora narekujejo različne velikosti pomnilnikov in različne kvalitete govora. Razmerja med temi parametri prikazuje diagram na sliki 3. [1].

Analiza in sinteza govora

Že prej sta bila omenjena dva osnovna načina delovanja analizatorjev in sintetizatorjev govora. Prvega imenujemo "kodiranje signalne krivulje" in kot že naziv pove, težimo pri tem načinu k čimbolj natančni reprodukciji govornega signala. Neobčutljivost takšnih sistemov na številne govorčeve lastosti in na vpliv šuma ter manjša kompleksnost izvedbe je dosežena z večjim pretokom podatkov (bit rate). Kodirniki signalne krivulje se lahko optimizirajo glede pretoka podatkov ter se ob upoštevanju statističnih rezultatov pri obdelavi govornih vzorcev, še bolj prilagodijo karakteristikam govornega signala.

Drugi način kodiranja temelji na skopem opisu parametrov govora, ki izhajajo iz ugotovitev o tem, kako govor nastaja. Takšne govorne sisteme imenujemo vokoderji (voice coder). Predpostavka je, da je mehanizem nastajanja govora razdeljen na dva ločena dela: izvor zvoka in govorni trakt, ki oblikuje zvok v glas. Nadalje, da izvor zvoka oddaja periodičen zven in šum, ki ga ustvarja zračni tok ter, da govorni trakt lahko simuliramo s spremenljivimi filtri. S številnimi meritvami lahko dosežemo optimalne vrednosti kmilnih parametrov in s tem tudi dobro kvaliteto govora, kar v praksi ni vedno izvedljivo. Vokoderji običajno generirajo manj kvaliteten (sintetični) govor ob minimalnem pretoku informacij (pod 4.8 Kbitov/s).

Področje med kodirniki signalne krivulje in vokoderji predstavlja prostor za raziskovanje in implementiranje kombiniranih postopkov za generiranje govora z izkoriščanjem dobrih lastnosti obeh omenjenih načinov.

KODIRNIKI SIGNALNE KRIVULJE

Metode za kodiranje signalne krivulje lahko delimo tudi na časovne in frekvenčne metode. Pri časovnih metodah je govorni signal obravnavan kot enojni signal na celem frekvenčnem območju in se postopki kodiranja razlikujejo glede na stopnjo prilagajanja signalu. Frekvenčne metode delijo signal na nekaj frekvenčnih komponent, ki jih nato ločeno kodirajo. Frekvenčne metode imajo to dobro lastnost, da je število bitov, ki je na razpolago za kodiranje ene frekvenčne komponente deljivo med komponentami glede na potrebe. Tako se frekvenčna komponenta, ki ni zastopana (ali je zastopana pod določeno mejo) sploh ne kodira.

Časovne metode

Neposredno digitaliziranje in shranjevanje govornega signala je logičen prehod iz analognega na digitalen sistem. Govorni signal se obravnava kot vhodni napetosno spremenjiv signal. Sistem (v našem primeru računalnik) posname govorno krivuljo na ta način, da periodično vzorči napetosti signala s pomočjo A/D pretvornika. Postopek, ki poljubni vrednosti vhodnega signala priredi najbližjo diskretno vrednost imenujemo kvantizacija. Vhodni signal je navzgor frekvenčno omejen z Nyquistovo frekvenco fN. Za zagotovitev sprejemljive kvalitete govora mora biti ta frekvenca med 3 in 4 KHz. Hitrost vzorčenja mora biti najmanj 2 x fN vzorcev na sekundo (Nyquistova hitrost) [3]. Vsak vzorec ima standardno dolžino 8 bitov, torej 2^8 možnih vrednosti s katerimi diskretno opisuje amplitudo vhodnega signala. Opisan postopek imenujemo impulzna modulacija PCM (puls-code modulation).

Pri kodirnih postopkih se kot merilo, uporablja podatek o pretoku informacije (bit rate). Ta, pri sistemih za prenos podatkov opisuje zasedenost prenosne linije, pri sistemih za shranjevanje podatkov pa opisuje potrebno velikost pomnilnika in je pri PCM metodi enak 2 x fN x B. Zmanjševanje te vrednosti je možno z ožanjem frekvenčnega območja vhodnega signala (govora) in zmanjšanjem števila bitov za posamezni vzorec. Frekvenčno območje ne sme biti pod 3 KHz, ker bi bila s tem močno znižana kvaliteta govora, če je vzoreo krajši kot 11 bitov pa se zmanjša učinek kvantizacije. Pri mejnih vrednostih obeh faktorjev znaša pretok informacije okrog 70 Kbitov/s. Primer uporabe PCM metode pri British Telecom govornem sistemu za telefonska obvestila kaže na obsežnost uporabljenega pomnilnika [1]. Besednjak 250-tih besed zaseda približno 6 Mbitov pomnilnika pri pretoku informacije 100 Kbitov/s. Pri tem je potrebno poudariti, da je reproduciran govor kristalno čist, teoretično kvaliteten tako kot digitalni posnetek govora, katerega karakteristike so splošno znane.

Velikost pomnilnika in s tem cena se poizkuša zmanjšati z novimi metodami ali z modifikacijami že znanih metod. Pretok informacije se zmanjša z zgoščanjem podatkov. PCM govorni signal vsebuje veliko odvečne informacije, posebno v področjih, kjer se

krivulja govora počasi spreminja. Zgoščevanje podatkov dosežemo z različnimi postopki kvantizacije govornega signala, ki se med seboj razlikujejo po doseženi stopnji predikcije: časovno spremenljiv korak kvantizacije, ki spremlja časovno spreminjanje vhodnega signala, logaritemska porazdelitev diskretnih stanj (LOG-PCM) ter razne oblike diferencialne kvantizacije (delta modulacija DM, prilagodljiva delta modulacija ADM, diferencialna impulzna modulacija DPCM in prilagodljiva diferencialna impulzna modulacija ADPCM).

Če se pri vzorčenju z Nyquistovo frekvenco shranjujejo podatki, ki pomenijo razliko med amplitudami dveh zaporednih vzorcev govorimo o diferencijalni PCM (DPCM).

Izboljšava PCM je tudi prilagodljiva DPCM (ADPCM), ki omogoča dobro kvaliteto govora pri manjšem pretoku informacije. Bistvo metode je dinamično spreminjanje postopka kvantiziranja, ki sledi dinamiki analognega govornega signala. Pri tem se število bitov pri vzorčenju zmanjša iz 12 pri PCM na 3-4 bite. Postopek spreminjanja kvantizacije je prilagojen spremembam govornih krivulj in ni uporaben za ostale vrste signalov. Tudi metoda ADPCM je integrirana v mikrovezjih različnih proizvajalcev [7].

Pri delta modulaciji, kot posebni obliki diferencijalne kvantizacije,se analogni vhodni signal vzorči in preoblikuje v enobitno informacijo. Ta opisuje smer spremembe nivoja signala med dvema vzorcema (prvi odvod). Pri reprodukciji se shranjeni vzorci integrirajo, kar sestavi ustrezen približek vhodnega signala [4].

Kvantizacijska napaka je definirana kot razlika med vzorcem izvirnega signala in signala, ki nastane po kvantizaciji in je pri delta modulaciji višja kot pri diferencijalni PCM zaradi enobitne informacije ob vzorčenju. Ta vrednost se pri delta modulaciji znanjšuje z višanjem frekvence vzorčenja, ki v tem primeru ni omejena z Nyquistovo frekvenco. Z dodajanjem integratorja pri kvantizatorju, se lahko spreminja ritem vzorčenja s čimer se doseže boljše spreljanje vhodnega signala. To metodo imenujemo prilagodljiva delta modulacija (adaptive delta modulation ADM).

Z dodatnim integratorjem, množilnikom in nekaj logike dobimo boljšo metodo, ki se imenuje delta modulacija s kontinuirano spremenljivim naklonom (continously variable slope delta modulation CVSD). Pri tej metodi se zniža pretok informacije na 16 Kbitov/s. Na tržišču obstajajo CVSD koderji / dekoderji v integriranem vezju raznih proizvajalcev kot so: MC3417 (Motorola) [5], FX-209 (Consumer Microcircuits of America) [6] in HC-55516 (Harris Semiconductor).

Metoda zgoščanja govornega signala uporablja kombinacijo opisanih metod z novimi ugotovitvami, ki so izšle iz analize govornih signalov in posebnosti našega dojemanja zvoka. Govorni signal je razdeljen na nekaj karakterističnih frekvenčnih področji (pitch periods). Pri kodiranju izbira računalnik med krivuljami takšne, ki so simetrične na sredino (zrcalne) in shranjuje samo informacije o eni polovici krivulje. Nato poišče krivulje z majhnimi amplitudami v prvi in zadnji četrtini posameznega frekvenčnega območja in te dele nadomesti s "tišino" (half period zeroing). Potopek se konča s kodiranjem vhodnega signala s prilagodljivo delta modulacijo.

Sinteza se začne s tihim področjem, nadaljuje se s četrtino periode govora, ki je shranjen v pomnilniku kot rezultat delta modulacije, ter z reprodukcijo istih podatkov v obratnem vrstnem redu. Zadnja četrtina pa je zopet tiho področje. Narava govornega signala dopušča, brez opazne popačitve zvoka, tri ali štirikratno zapovrstno ponovitev shranjenega izhodnega signala za nezveneče glasove, ali tudi osemkratno ponovitev za zveneče glasove. Število ponovitev signala, čas trajanja vmesnih tihih področji, izbiro med moško ali žensko višino govora in ostala navodila posredujejo sintetizatorju ukazne besede. Z opisanim postopkom se zmanjša pretok informacije na 1 Kbit/s pri zadovoljivi kvaliteti govora.

Frekvenčne metode

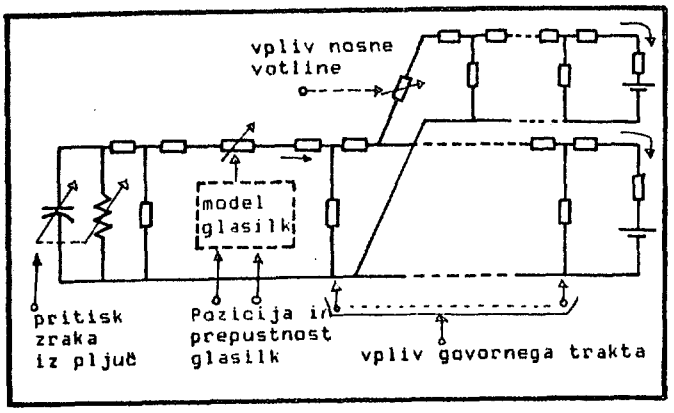
Tako kot pri časovnih metodah, tudi frekvenčne metode se medseboj razlikujejo po kompleksnosti izvedbe in stopnji predikcije. Iz množice različnih metod bomo omenili le dve, pasovno kodiranje in transformacijsko kodiranje.

Pri pasovnih kodirnikih je frekvenčno območje govornega signala, s pomočjo množice pasovnih filtrov, razdeljeno običajno na štiri do osem frekvenčnih pasov. Proces kodiranja se potem nadaljuje ločeno za vsak pas, po eni od zgoraj opisanih metod (npr. APCM). Ob dekodiranju digitalnega zapisa se dekodirani signali v posameznih pasovih združijo v enojni govorni signal. Dobra lastnost opisane metode je v tem, da kvantizacijske napake ostajajo znotraj posameznih pasov in se ne prenašajo (prištevajo) napakam v ostalih pasovih. Obenem ta metoda omogoča zmanjševanje kvantizacijske napake z uporabo različnih kvantizacijskih korakov v posameznih pasovih.

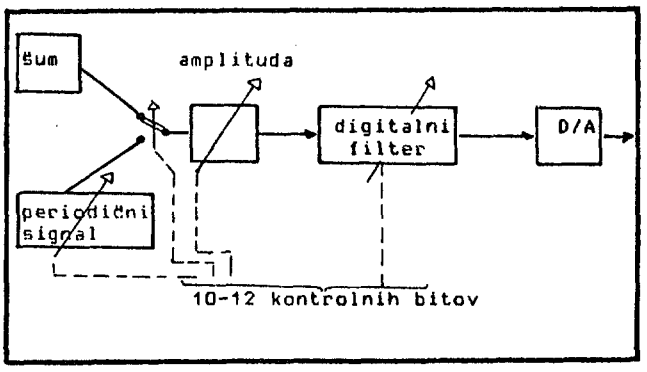
Bolj kompleksna je metoda prilagodljivega transformacijskega kodiranja. Valovna oblika govornega signala se segmentira; nato se nad segmentom signala izvede ena od diskretnih časovno-frekvenčnih transformacij (za govorni signal je najprimernejša diskretna cos transformacija), katere rezultat je množica transformacijskih koeficientov. Koeficienti se potem ločeno kvantizirajo. Rekonstruiranje govornega signala poteka v obratnem vrstnem redu, razumljivo z uporabo inverzne transformacije. Boljše rezultati se dosežejo z upeljavo dodatne informacije o dinamiki govora znotraj posameznega segmenta. Ta, t.i. "stranska informacija", ki predstavlja spektralni nivo znotraj posameznega segmenta, omogoča prilagodljivost opisane metode.

VOKODERJI

Do sedaj omenjene metode sodijo v skupino metod, ki temeljijo na analizi valovne oblike govornega signala in pri katerih ni pomembno nastajanje temveč dojemanje zvoka. Obstajajo pa tudi metode, katerih osnova so elektronski modeli posameznih delov govornega trakta (grlo, nosni kanali, glasilki, ustna votlina itd ...). Glas nastaja s filtriranjem zvenečih (generator čistega zvena) in nezvenečih (generator šuma) zvokov. Prenosne funkcije filtrov ponazarjajo delovanje posameznih delov govornega trakta, ki so udeleženi pri nastanku glasu (Slika 4).



Slika 4: Digitalni model govornega trakta.



Slika 5: LPC generator govora.

Vokoderji se med seboj razlikujejo po obliki parametrov s katerimi je opisan govorni trakt. Obstajajo različni tipi parametrov kot so: kratkotrajni amplitudni spekter govornega signala na posameznih frekvencah (kanalni vokoderji); koeficienti linearne predikcije, ki opisujejo spektralno ovojnico (LPC vokoderji); podatki o frekvenci ob glavnih resonančnih frekvencah govornega trakta (formantni vokoderji); vzorci kratkotrajnih avtokorelacijskih funkcij govornega signala (avtokorelacijski vokoderji); koeficienti množice ortonormiranih funkcij, ki aproksimirajo govorno krivuljo (ortogonalno funkcijski vokoderji) in številni drugi tipi.

Opisali bomo dva od naštetih tipov. Prvega (LPC vokoder) uvrščamo med časovne, tako kot avtokorelacijske in ortogonalno funkcijske, za razliko od formantnih in kanalnih, ki sodijo med frekvenčne vokoderje.

LPC vokoderji

Osnovna ideja linearne predikcije je predpostavka, da če ugotovimo, koliko informacije vsebujejo o i-tem vzorcu, predhodni vzorci, pričakujemo, da bomo lahko z njimi do določene mere napovedali i-ti vzorec. Govor se analizira tako, da se analogni govorni signal najprej vzorči s hitrostjo med 8 in 10 KHz. Iz analize vzorcev se določijo vrednosti krmilnih parametrov digitalnega filtra, določa se ali je zvok zvenek ali ni in osnovna frekvenca ter amplituda govora.

Število krmilnih parametrov digitalnega filtra je odvisno od željene kvalitete govora in znaša med 10 pri slabši kvaliteti do 12 pri dobri kvaliteti reproduciranega govora. Pretok informacije je pri tej metodi med 3000 in 1000 bitov/s, kar se doseže z dodatnim krčenjem shranjenih podatkov. V primerih, ko se zahteva samo razumljivost govornega sistema pade pretok informacij med 1000 in 20 bitov/s.

Sinteza govora z LPC metodo je razdeljena na dva osnovna dela: krmiljenje vhodnega signala in krmiljenje digitalnega filtra (Slika 5). Vhodni signal (zvok) ima dva vira. Generator psevdo naključnega šuma omogoča nastajanje nezvenečih glasov. Generator čistega zvena (pravilni periodični signal) omogoča nastajanje zvenečih glasov npr. samoglasnikov. Govor nastane tako, da se vhodni signal (šum ali periodični signal določene frekvence) določene amplitude oblikuje skozi N-polni LPC digitalni filter, ki predstavlja model človeškega govornega sistema.

Formantni vokoderji

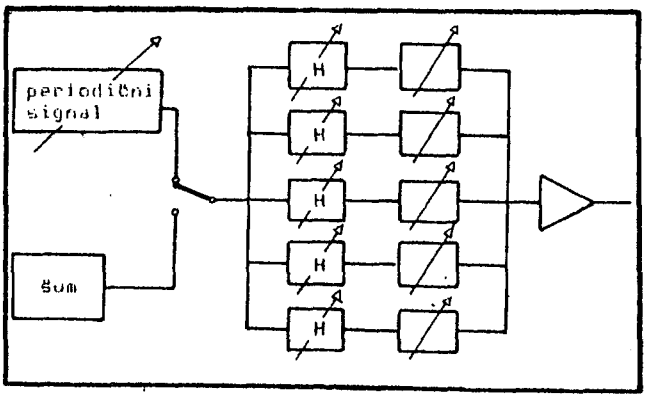
Formantni generatorji govora so po materialni opremi zelo podobni LPC sintetizatorjem (Slika 6). Razlika je v tem, da se pri tej metodi uporabljajo podatki o poziciji središča in pasovnih širin formantnih frekvenc, ki so rezultat predhodne analize govora. Formanti so področja z visoko govorno energijo [3], ki so prisotna na različnih frekvencah govornega signala in so v bistvu resonančne frekvence govornega trakta.

F0	50	...	250 Hz
F1	200	...	900 Hz
F2	600	...	2500 Hz
F3	1500	...	3000 Hz
F4	3000	...	4000 Hz
F5	.....		
šum	3000	...>8000	Hz

Tabela 2: Področja formantnih frekvenc za moški glas.

Običajno so tri ali štiri formantne frekvence, ki se gibljejo za moški glas med vrednostmi, ki so prikazane v tabeli 2. Formantne frekvence ženskega glasu so približno 25% višje. Pasovne širine posameznih formantnih frekvenc se določajo algoritmično pri sintezi, ker v splošnem, natančno določanje pasovnih širin ne upliva direktno na kvaliteto generiranega govora.

Opisana metoda omogoča še manjši pretok informacije, ki se giblje med 600 in 800 bitov/s za kvaliteten govor.



Slika 6: Formantni generator govora.

Zaključek

Posebno področje pri govorni komunikaciji med strojem in človekom predstavljajo sistemi za pretvorbo teksta v govor, ki se pogosto uporabljajo pri govoređih terminalih ali pisalnih strojih in so v veliko pomoč vidno prizadetim osebam. Pri teh sistemih predstavlja posebnost dovolj bogata banka znanja o prevajanju pisanih besed v govor [16, 17]. Običajno se vhodni tekst z računalnikom prevede v foneme ali zloge, ki se potem s pomočjo ene od zgoraj omenjenih metod oblikuje v govor.

V zadnjih letih se pri generiranju govora uporabljajo splošni, ali posebno za govor zgrajeni, hitri signalni procesorji. Zaradi hitrosti procesiranja omogočajo izvajanje algoritmov s katerimi simulirajo posamezne dele analognih vezij, ki gradijo LPC ali formantne digitalne filtre [18,19,20].

Tako LPC kot formantne generatorje govora lahko že nekaj let kupimo na tržišču v obliki integriranih vezij, kompletov integriranih vezij na tiskani plošči ali govornih sistemov. Lahko naštejemo nekaj proizvajalcev: American Microsystems (LPC 1400 bitov/s), General Instruments (LPC vezja in plošče), Centigram (PWC metoda podobna LPC, 4800 bitov/s), EG&G Reticon (signal procesorji), E-Systems (LPC, 2400 bitov/s), Mimic Electronics (vezja in plošče), ITT Semiconductors, Matsushita, National Semiconductor, Nippon Electronic , Sharp, Telesensory Systems, Texas Instruments, Toshiba, Triangle Digital Services (plošče, 650-6500 bitov/s), Votrax, Yahara (govorni sistemi) in mnogi drugi.

Področja uporabe opisanih govornih sistemov so različna. Izbira metode in postopka je odvisna od zahtevane kvalitete in cene izdelka. Znani so govorni sistemi za računalniško generiranje ožičevalnih list, za telefonično sporočanje različnih podatkov in informacij o telefonskih naročnikih in vremenskih podatkih ter podatkih o cenah proizvodov ter za sporočanje podatkov o letalskih poletih. Poleg velikih sistemov obstajajo in še nastajajo neštete manjše aplikacije ob privatnih računalnikih, za alarmna sporočila, za navodila pri uporabi različnih proizvodov in podobno.

V prispevku smo podali pregled zgodovine in sodobnega stanja na področju računalniške sinteze govorjenega naravnega jezika.

Glede na aktualnost in uporabnost področja (pri tem naj zlasti omenimo generiranje govora iz odprtega teksta/sporočil/podatkov) in glede na dejstvo, da bo moral vsak (vsaj majhen) narod sam poskrbeti za osnovne raziskave - brez katerih ni mogoča kvalitetna aplikacija današnjih tehničnih možnosti - na področju morfofonoloških značilnosti svojega jezika, je smiselno napovedati naslednji prispevek s sledečo vsebino:

- pregled dosedanjega dela in doseženih rezultatov pri nas na področju matematičnih in tehničnih osnov (materialna in programska oprema) na področju sinteze govora;
- oris pristopa k osnovnim raziskavam posebnosti slovenščine, ki jih moramo obvladati, če naj naši sintetizatorji govorijo slovensko in ne ameriško slovenščino;
- pristop k ustvarjanju ustreznega materialnega in programskega okolja za osnovne in aplikativne raziskave sinteze slovenščine.

Literatura*

[1] Voice Input/Output sistem and devices, Electronic Engineering, May 1982, page 76 - 101

[2] James L. Flanagan : Computers that Talk and Listen: Man - Machine Comunication by Voice, Proceedings of the IEEE, April 1967, page 405 - 415

[3] Lawrence R. Rabiner, Ronald W. Schafer: Digital Tehniques for Computer Voice Response Implementations and Applications, Proceedings of the IEEE, April 1967, page 416 - 432

[4] Raymond Steele : Chip delta modulators revive designer's interest, Electronics, October 13, 1977, page 86 - 93

[5] Earle West : IC trio simplifies speech synthesis, Electronic Design, October 28, 1982, page 175 - 178

[6] FX-209 Analogue / Digital Converter, Product Information,Consumer Microcircuits Limeted

[7] Steve Carcia : Use ADPCM for Highly Intelligibile Speech Synthesis, Byte, June 1983, page 35 - 49

[8] Cecil H. Coker : A Model of Articulatory Dynamics and Control, Proceedings of the IEEE, April 1967, page 452 - 474

[9] Computing Power Boosts Speech Quality, Digital Design, May 1982, page 44 - 50

[10] Gene Helms & Steve Petersen : Portable speech development system creates linear predictive codes, Electronics,September 8, 1982, page 151 - 156

[11] Speech synthesis: device and applications, Electronic Engineering, January 1981, page 41 - 57

[12] Roger J. Godina : Voice input output : a special report, Electronics, April 21, 1983, page 126 - 143

[13] R. Pickvace : Speech synthesis; the new frontiers, Electronic Engineering, Juli 1980, page 37 - 52

[14] Kathryn Fons and Tim Gargaliano : Articulate Automata: An Overview of Voice Synthesis, Byte, February 1981, page 164 - 187,

[15] Noriko Umeda : Linguistic Rules for Text - to - Speech Syntheses, Proceedings of the IEEE, April 1967, page 443 - 451

[16] A. Yatagai : Limited and unlimited vocabulary speech synthesis systems, Electronic Engineering, November 1980, page 31 - 41

[17] Jonathan Allen : Synthesis of Speech from Unrestricted Text, Proceedings of the IEEE, April 1967, page 433 - 442

[18] Edward Bruckert, Martin Minow and Walter Tetshner : Three-tiered software and VLSI aid developmental system to read text aloud, Electronics, April 21, 1983 , page 133 - 138

[19] M.E.Hoff and Matt Townsend : Single-chip NMOS microcomputer processes signals in real time, Electronics, March 1, 1979, page 105 - 110

[20] M.E.Hoff and Waillace Li : Software makes a big talker out of the 2920 microcomputer, Electronics, January 31, 1980, page 102 - 107

[21] Veljko Zgaga : Sinteza govora u mikroračunarskim sistemima, Zbornik Informatica 77, Bled 1977, 2 212

[22] James C. Anderson : An Extremely Low-Cost Computer Voice Response System, Byte, February 1981, page 36 - 43

---

* Zaradi pomanjkanja domačih prispevkov o tej temi je podan obsežnejši spisek tujih prispevkov, ki obravanavajo področje sinteze govora.

[23] Tomihiro Matsumura: Future Microprocessor Trends, Information Processing 83, R.E.A.Mason(ed.) page 213 - 217
[24] Bruce LeBoss: Speech I/O is making itself heard, Electronics, May 22, 1980, page 95 - 105
[25] Michael W. Hutchins and Lee Dusek : How Vocabulary is Generated Determines Speech Quality, Computer Design, February 1984, page 89 - 96
[26] Single silicon chip synthesizes speech in $50 learning aid, Electronics, June 22, 1978
[27] James L. Flanagan, Manfred R. Schroder, Bishnu S. Atal, Ronald E. Crochiere, Nuggehally S. Jayant, Jose M. Tribolet : Speech Coding, IEEE Transaction on Communications Vol. COM-27, No. 4, April 1979, page 710 - 735
[28] R.E.Crochiere,S.A.Webber and J.L.Flanagan Digital Coding of Speech in Sub-bands, The Bell System Technical Journal, Vol 55 No 8, October 1976, page 1069 - 1085
[29] D.J.Goodman,B.J.McDermot and L.H.Nakatani: Subjective Evaluation of PCM Coded speech, The Bell System Technical Journal, Vol 55 No 8, October 1976, page 1087 - 1109
[30] N.S.Jayant : Pitch-Adptive DPCM Coding of Speech With Two-Bit Quantization and Fixed Spectrum Prediction, The Bell System Technical Journal, Vol 56, No 3, March 1977, page 439 -454
[31] L.R.Rabiner, C.E.Schmidt and B.S.Atal : Evaluatin of Statistical Approach to Voiced - Unvoiced - Silence Analysis for Telephone-Quality Speech, The Bell System Technical Journal, Vol 56, No 3, March 77 page 455 -482
[32] D.J.Goodman, C.Scagliola, R.E.Crochiere, L.R.Rabiner and J.Goodman : Objective and Subjective Performance of Tandem Connections of Waveform Coders with an LPC Vocoder, The Bell System Technical Journal, Vol 58, No 3, March 1979, page 601 - 629.
[33] J.J.Dubnowski and R.E.Crochiere: Variable Rate Coding of Speech, The Bell System Technical Journal, Vol 58, No 3, March 79 page 577 - 600
[34] N.S.Jayant and S.W.Christensen : Adaptive Aperture Coding for Speech Waveforms - I, The Bell System Technical Journal, Vol 58 No 7, September 1979, page 1631 -1645
[35] J.L.Flanagan, J.D.Johnston, J.W.Upton : Digital Voice Storage in a Microprocessor, IEEE Transac. on communicat. Vol COM-30, No 2, February 1982, page 336 - 345

INFORMATICA 2/85

UPORABA DIGITALNEGA INKREMENTALNEGA DAJALNIKA KOTA V MIKRORAČUNALNIŠKEM KRMILJU INDUSTRIJSKEGA ROBOTA

S. ZORMAN

UDK: 681.3:51

INSTITUT JOŽEF STEFAN, LJUBLJANA

Članek podaja princip delovanja digitalnega inkrementalnega dajalnika kota, ter opisuje potrebno dodatno logiko, pri njegovi uporabi v mikroračunalniškem krmilju industrijskega robota.

APPLICATION OF A DIGITAL INCREMENTAL ENCODER ON A MICROCOMPUTER CONTROL SYSTEM FOR INDUSTRIAL ROBOTS

This paper describes the principle of operation of a digital incremental encoder and the supplementary logic which is necessary for its application on a microcomputer control system for industrial robots.

1. UVOD

Linearne in rotacijske premike v zglobih industrijskega robota lahko merimo z analognimi ali digitalnimi merilniki. Doslej smo pri robotih, ki smo jih razvili na Institutu Jožef Stefan, uporabljali za merjenje lege analogne servopotenciometre.

Merjenje lege s servopotenciometri ima svoje dobre in slabe strani. Dobro je, da je informacija o legi dana absolutno, kar pomeni, da poznamo pravo lego tudi takoj po zagonu sistema. Po drugi strani je analogni signal servopotenciometra podvržen motnjam. Na informacijo o legi vpliva linearnost potenciometra, potrebna pa je tudi analogno/digitalna (A/D oziroma digitalno/analogna (D/A) pretvorba, če naj bi izmerjeno lego primerjali z želeno.

Zato, da bi se izognili vplivu motenj in potrebi po A/D oziroma D/A pretvorbi, smo namesto servopotenciometrov uporabili kot merilnike lege digitalne inkrementalne dajalnike kota. Uporaba digitalnih inkrementalnih dajalnikov kota ima za posledico težave z zagonom sistema, ker ob zagonu nimamo veljavne informacije o legah segmentov robota. S primernim postopkom ob zagonu sistema postane ta pomanjkljivost nepomembna. Prednosti, ki jih nudi ta način merjenja lege, so zmanjšana verjetnost vpliva motenj, izognemo se potrebi po A/D oziroma D/A pretvorbi in poveča se točnost meritev.

Namen članka je, da nas seznani z možnostjo uporabe digitalnega inkrementalnega dajalnika kota kot merilnika lege v mikroračunalniškem krmilju industrijskega robota.

2. PRINCIP DELOVANJA DIGITALNEGA INKREMENTALNEGA DAJALNIKA KOTA

Digitalni inkrementalni dajalnik kota (Sl. 1), v nadaljevanju dajalnik kota, je sestavljen iz ohišja, v katerem je v ležaj vpeta os, na katero je pritrjen disk z režami.

V ohišju je poleg že naštetega še svetlobni vir, fiksno nameščen zaslon z režami ter elektronika s svetlobnimi detektorji.

Disk, ki je pritrjen na os je izdelan tako, da je za svetlobo iz svetlobnega vira nepropusten povsod, razen na svojem obodu. Na obodu diska so izdelane reže, skozi katere prodira svetloba iz svetlobnega vira. Reže imajo radialno smer in so vse enake dolžine, razen ene, ki je podaljšana. Svetlobni vir je nameščen na eni strani diska, na drugi strani diska pa se nahaja zaslon z režami in za njim svetlobni detektorji.

Reže na zaslonu so razporejene v dve skupini in eno osamljeno režo. Zaslon je nameščen tako, da se reže na zaslonu precizno ujemajo z režami na disku. Skupini rež na zaslonu sta med seboj tako zamaknjeni, da če zavrtimo disk v tak položaj, da se reže prve skupine ujemajo z režami na disku, so reže druge skupine samo do polovice ujete z režami na disku.

Na vsakega od detektorjev svetlobe pada svetloba le skozi eno od skupin oziroma osamljeno režo zaslona. Tako dobimo tri električne signale (Sl. 2), ki so potrebni, da je mogoče slediti legi osi dajalnika kota.

S pomočjo vseh treh signalov dajalnika kota in zunanje dodatne logike je mogoče slediti legi osi dajalnika kota.

3. DODATNA LOGIKA

Naloga dodatne logike je, da iz signalov dajalnika kota izlušči informacijo o legi in jo posreduje računalniku krmilja na njegovo zahtevo. Bločno shemo dodatne logike prikazuje slika 3.

Signali dajalnika kota vstopajo v blok z vhodnimi ojačevalniki in invertorji (blok 1.), ki so potrebni zato, da je mogoče preko enake dodatne logike priključiti dajalnike kotov različnih proizvajalcev. Elementi tega bloka nam dobro služijo tudi za definiranje pozitivne oziroma negativne smeri vrtenja.

Kakšen najmanjši premik je potreben, da se nam spremeni stanje signalov dajalnika kota, je odvisno od delitve na njegovem disku. Pri tem stanje smatramo kombinacijo logičnih nivojev izhodnih signalov dajalnika kota, V primeru, da ima dajalnik kota 2540 rež na obodu diska, tedaj je njegovo os v idealnem primeru potrebno zasukati za 0.035 kotne stopinje, da bi se njegovo izhodno stanje spremenilo. S tem je podana tudi maksimalna ločljivost premika segmenta robota, ki je spet z danim dajalnikom kota.

Zato, da bi se navedena ločljivost dosegla, je potrebno detektirati vsako spremembo stanja dajalnika kota. Ob vsaki detektirani spremembi je potrebno prožiti števni impulz za števec in določiti smer štetja, ki mora ustrezati smeri vrtenja osi dajalnika kota.

Vezje z vlogo generatorja števnih impulzov (blok 2) je preprosto logično vezje, sestavljeno iz dveh EXOR vrat in zakasnilnega RC člena (sl.4). Na isti sliki je prikazana tudi medsebojna odvisnost vseh signalov vezja.

Vhodna signala za generator števnih impulzov sta signala A, B, ki imata obliko vlakov imulzov, če se os dajalnika kota vrti. Vezje na sliki 4. ob vsaki spremembi logičnih nivojev enega ali drugega vhodnega signala, na izhodu generira kratek impulz, katerega širina je definirana s časovno konstanto R1C1.

Naloga števca (blok 7) je, da ob prihajajočih števnih impulzih svojo vsebino povečuje ali zmanjšuje po ena, odvisno od smeri vrtenja osi dajalnika kota.

Informacija o smeri, ki jo dobiva števec, mora biti točna le tedaj, ko pride do števca števni impulz, sicer pa to ni nujno potrebno. Informacijo o smeri gibanja osi dajalnika kota izlušćimo iz istih dveh signalov, ki služita tudi za generiranje števnih impulzov. Signala sta fazno zamaknjena. Kateri izmed obeh prehiteva drugega je odvisno od smeri vrtenja. Vezje, ki izlušči informacijo o smeri vrtenja (blok 3), je še preprostejše kot tisto za generiranje števnih impulzov. Prikazano je na sliki 5. Edina zahteva v zvezi s tem vezjem je, da sta R2 in C2 izbrana tako, da je R2C2 konstanta tega vezja večja od R1C1 konstante vezja za generiranje števnih impulzov.

Na sliki 5. je poleg vezja podan tudi časovni diagram signalov iz katerega je razvidno, da se števni impulzi v primeru vrtenja v eno smer generirajo, ko je signal smeri SM v logično visokem stanju. Ko se smer vrtenja obrne, se zamenja tudi logični nivo, pri katerem se generirajo številni impulzi. Signal SM določa smer štetja števca. Z invertiranjem enega od obeh vhodnih signalov dosežemo tudi invertiranje signala SM. S tem je torej mogoče na enostaven način določiti katera smer vrtenja pomeni vrtenje v pozitivnem oz. negativnem smislu.

Do sedaj opisano vezje skupaj z dvosmernim števcem, zadostuje za poznavanje lege glede na nek začetni položaj, ki pa v splošnem ob zagonu sistema ni definiran. Pri robotu je potrebno poznati lege posameznih segmentov glede na referenčne položaje. V tem primeru se izkaže za koristnega tretji signal dajalnika kota - C, signal referenčne lege.

Ob zagonu sistema moramo poskrbeti, da gredo vsi segmenti robota od ene do druge skrajne lege. S tem zagotovimo, da gredo tudi skozi svoje referenčne lege. Blok 4. predstavlja del dodatne logike, ki vrednost števca ob prehodu skozi referenčno lego postavi na nič. S tem dosežemo, da od zagona naprej poznamo lege posameznih segmentov glede na njihove referenčne lege.

Zato, da računalnik lahko pride do veljavnih podatkov o legi v vsakem trenutku, je potrebno med števec in podatkovno vodilo računalnika postaviti še register (blok 5). Na ta način se prepreči spreminjanje podatka o legi v času čitanja, hkrati pa se omogoči neprekinjeno delovanje števca.

Za prenos podatka med števcem in registrom poskrbi osveževalnik vsebine registra (blok 5). Le-ta poskrbi, da se na zahtevo računalnika prenese v register vsebina števca, vendar šele tedaj, ko je vsebina števca stabilna. Osveževalnik vsebine registra v primeru, ko se zahteva prenos iz števca v register v času, ko je prisoten števni impulz, prenos zakasni za toliko, da se vsebina števca lahko ustali.

Dajalnik kota in dodatna logika predstavljata za računalnik periferno enoto. Kot taka mora imeti svoj naslov, preko katerega je računalniku dosegljiva. Dekodirna logika (blok 8) je sestavni del dodatne logike, ki služi dekodiranju tega naslova in na ta način omogoča odzivanje dodatne logike na klice računalnika.

4. ZAKLJUČEK

Sam digitalni inkrementalni dajalnik kota ni zadosten za podajanje lege. Potrebno mu je dodati logiko, ki iz signalov dajalnika kota izlušči informacijo o legi in ji nadene številčno podobo.

Na Institutu Jožef Stefan smo potrebno dodatno logiko razvili in jo tudi uspešno preizkusili. Pri tem smo uporabljali dajalnik kota proizvajalca TELDIX model 702 SR - 2540. Uporabljen dajalnik daje 2540 impulzov na obrat, kar pomeni, da lahko zasuk osi dajalnika kota določimo na 0.035 kotne stopinje natančno, če predpostavimo, da je dajalnik idealen. Zaradi odstopanj od idealnih razmer, kar je posledica omejitev pri izdelavi diska in zaslona dajalnika, se točnost izmerjenega kota zmanjša na 0.053 kotne stopinje. 0.053 kotne stopinje je torej pogrešek s katerim lahko merimo kote med 0 in 360 stopinjami.

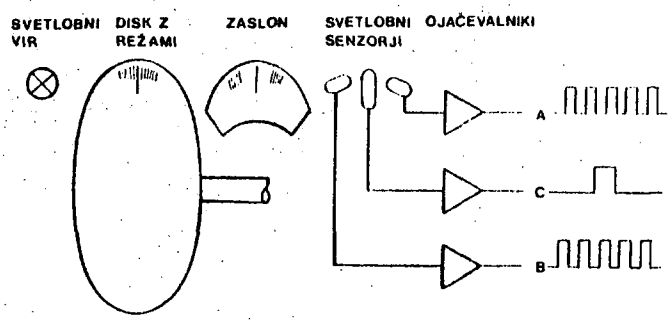
Za primerjavo si poglejmo še na kakšen pogrešek moramo računati v primeru, da za merjenje lege segmentov robota uporabljamo servopotenciometre. Vzemimo, da je električni kot potenciometra 260 kotnih stopinj in da imamo na voljo deset bitno A/D pretvorbo. Predpostavimo še, da sta potenciometer in A/D pretvorba idealno linearna in da ni nobenih motenj, ki bi lahko vplivale na A/D pretvorbo. Skratka vzemimo pod lupo idealen primer. Pogrešek na katerega moramo računati je 0.254 kotne stopinje. Pri tem se moramo zavedati, da lahko s takim pogreškom merimo le kote, ki niso večji od 260 kotnih stopinj.

Enostaven račun nam pokaže, da je faktor izboljšanja že v primeru, če primerjamo realno pričakovan pogrešek merjenja z dajalnikom kota z idealiziranim pogreškom pri meritvi s potenciometrom, skoraj pet.

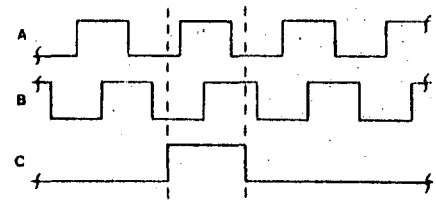
Uporaba dajalnika kota za določanje leg posameznih segmentov robota je le ena od možnih uporab . Drugo področje na katerem so dajalniki kota in dodatna logika zanje uporabni je nedvomno področje numerično kontroliranih strojev. Gotovo pa jih je mogoče s pridom uporabiti še kje.

5. LITERATURA

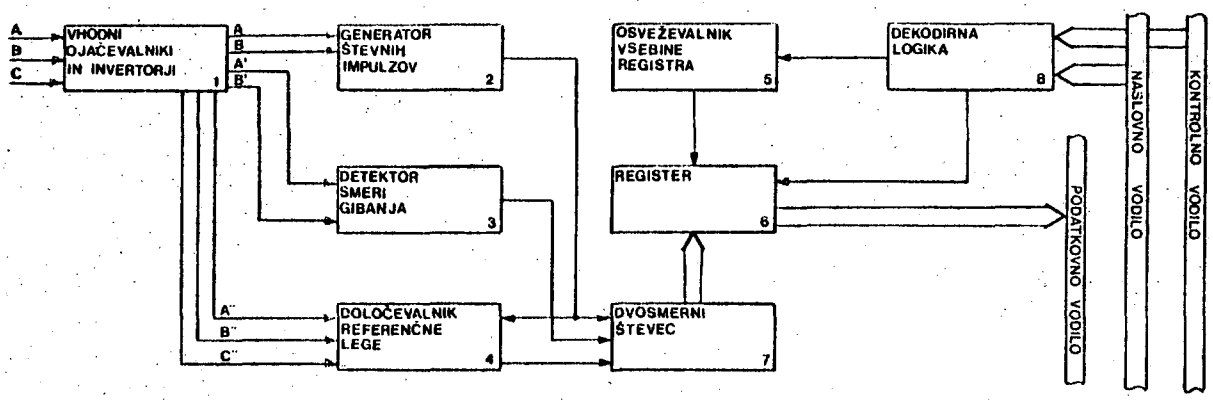
G. H. Woolvet
Transducers in Digital Systems,
IEE Control Engineering Series 3, 1977



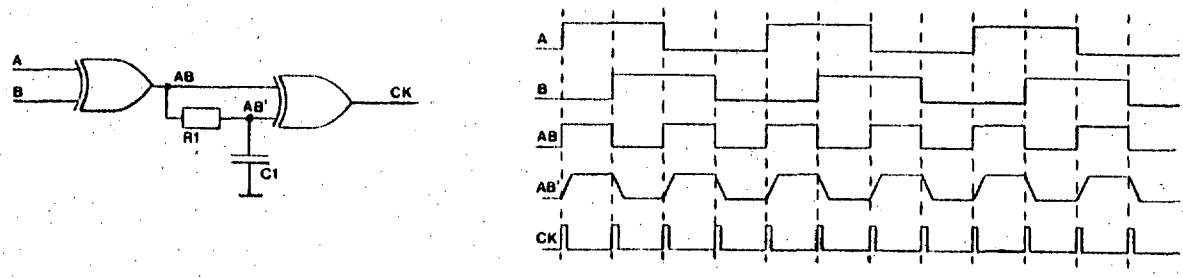
Sl. 1 Shematski prikaz dajalnika kota.



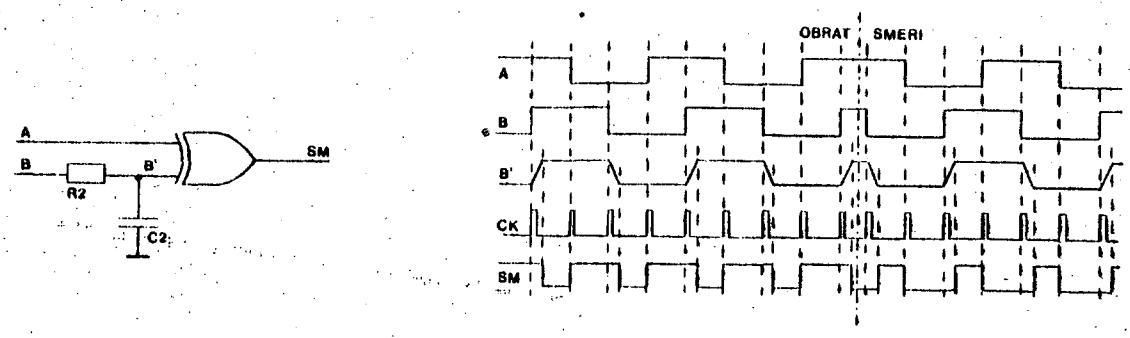
Sl. 2 Signali dajalnika kota.



Sl. 3 Bločna shema dodatne logike.



Sl. 4 Vezje generatorja števnih impulzov s časovnim potekom signalov.



Sl. 5 Vezje detektorja smeri gibanja osi dajalnika kota s časovnim potekom signalov.

INFORMATICA 2/85

PRIMJER DINAMIČKOG RASPOREDJIVANJA ZADATAKA U VIŠERAČUNARSKIM SISTEMIMA SA RADOM U STVARNOM VREMENU

KUKRIKA MILAN

ELEKTROTEHNIČKI FAKULTET
BANJALUKA

UDK: 681.324

SAŽETAK - Raspodijeljeni sistemi kod kojih je ostvareno dinamičko pridjeljivanje zadataka računalima mogli bi se u pogledu brzine odziva, iskoristivosti i cijene pokazati boljima od konvencionalnih sistema. Predložena konfiguracija sastoji se od skupa radnih računala i mreže komunikacijskih računala koja predstavljaju spregu radnih računala i linije za povezivanje sa drugim računalima. Sistem je koncipiran tako da se zahtjevi za izračunavanjima koji prevazilaze mogućnosti odredjenog računala, ili koja degradiraju njegove performanse, prenose ostalim slobodnim računalima u sistemu. Zadaću pronalaženja optimalnog dinamičkog pridjeljivanja zadataka, u cilju izjednačavanja opterećenja u sistemu, tada preuzima globalni raspoređivač. Globalni raspoređivač se sastoji od skupa identičnih algoritama koje izvode komunikacijska računala.

ABSTRACT - AN EXAMPLE OF DYNAMIC TASK SCHEDULING IN A REAL-TIME MULTICOMPUTER SYSTEMS. A distributed systems which supports the dynamic assignment of tasks between symetrical computers is offering a significant vantage in response speed, use and cost-efective performance than the conventional systems. This paper presents a distributed system as a collection of autonomous host computers networked together with a communication computers. System is designed so that the demands for computations which exceed the resource bounds of the computer, or which degrade its performance may be asigned to a another systems computers, It is the task of the global scheduler to find an optimal dynamic assignment of tasks, so that a load balancing in system is achieved. Global scheduler consists of a set of replicated algorithms, each residing and running in the communication computer.

1. UVOD

Glavne prednosti raspodijeljenih sistema u odnosu na druge računarske strukture, koji motiviraju daljnja istraživanja u oblasti njihovog projektiranja i izgradnje su modularnost, pouzdanost, propusnost sistema, brzina, efikasnost rada, mogućnosti proširenja, raspoloživost, prihvatljive cijene i druge. To su i osnovni ciljevi koje bi trebalo postići izgradnjom takvog sistema, a njihovo ostvarenje bitno ovisi od kvaliteta sistemske programske podrške.

S druge strane prednosti koje uvodi uporedno izvršavanje zadataka u raspodijeljenim sistemima ograničene su složenošću realizacije sistemske programske podrške. Mnoga rješenja koja su se pokazala dobrim kod multiprogramskih i višeprocesorskih sistema ne daju se jednostavno preslikati i na raspodijeljene sisteme.

Pretpostavke od kojih se mora poći pri kreiranju sistemske programske podrške su množina ciljeva (zahtjeva) i oskudnost sredstava potrebnih da bi se ti zahtjevi ispunili. Dijeljenje sredstava izmedju više procesa je posljedica situacije u kojoj se oskudijeva u sredstvima, jer nema dovoljno memorija i procesora za istovremeno izvršavanje svih pripravnih procesa.

Problem racionalne raspodjele sredstava može se rješavati samo u slučaju kada raspolažemo ocjenom (kriterijumom) koja omogućava da se u svakoj konkretnoj situaciji odabere optimalna varijanta raspodjele sredstava, pri kojoj su zadovoljena ograničenja postavljena sistemu, a vrijednost kriterija dostiže najpovoljniju veličinu. Pošto prilikom dodjele procesora procesima moramo da računamo sa nizom faktora različitog značaja, a izbor bi trebalo da se izvrši prema jedinstvenoj ocjeni, može se kao odgovarajuća kompleksna ocjena promatrati kriterijum I, koji uzima u obzir posebne ocjene sistema po pojedinim faktorima prema njihovom relativnom značaju. Takav kompleksan kriterijum može da bude izraz

$$I = f(I_1, I_2, \ldots I_n) = \sum_i a_i I_i$$

gdje su I_i ocjene prema pojedinim pokazateljima, a a_i težinski koeficijenti koji uzimaju u obzir značaj pojedinih pokazatelja.

Raspodijeljeni sistemi u odnosu na jednoprocesorske posjeduju još jednu klasu sredstava koje raspoređuje izvršni sistem, a to procesori. Ako se zahtijeva da se u stvarnom vremenu izvede n zadataka na m procesora (n > m) odluka o raspoređivanju zadataka po računalima, kao i odredjivanje redoslijeda njihovog izvršavanja u svakom od računala može bitno utjecati na performanse raspodijeljenog sistema.

Stoga se postavlja zadatak da se za svako zadano stanje sistema i okoline nadje i realizira takva raspodjela zadataka po procesorima pri kojoj će kriterijum efikasnosti funkcioniranja sistema poprimiti najpovoljniju vrijednost.

2. STRATEGIJE RASPOREDJIVANJA

Kod sistema sa radom u stvarnom vremenu pojedine aktivnosti (zadaci) odvijaju se pod uticajem okoline i njihov

redoslijed se ne može u potpunosti unaprijed predskazati. Zadaci u sistemu moraju stoga biti povezani na takav način da ispravno suradjuju u obavljanju predvidjenog posla u svim dinamičkim uvjetima koje nameće okolina. Stoga je za optimizaciju rada raspodijeljenih sistema od najveće važnosti pristup raspodjeli zadataka.

Kod sistema sa statičkim pridjeljivanjem zadataka funkcije pojedinih računala su unaprijed tačno odredjene. Time je i sistemska programska podrška jednostavnija, jer su jednostavnije i forme komunikacije. Medjutim, intuitivno možemo zaključiti da je takva struktura neefikasna, jer se može dogoditi da se u nekim računalima formira rep spremnih zadataka dok su druge neopterećene. Kod brze i nepredvidljive promjene sistemi sa strukturalnom neodredjenošću se pokazuju boljima od statičkih sistema.

Kod sistema sa dinamičkim pridjeljivanjem zadataka pojedini zadaci mogu biti izvršeni od više, ili čak od svih računala u sistemu. Sistemska programska podrška je bitno složenija nego u prethodnom slučaju, ali očekujemo poboljšanje u odnosu na prethodni pristup jer će zadatak kraće čekati na izvršenje, pošto će biti pridijeljen najmanje opterećenom računalu.

Početak izvršavanja svakog od zadataka zavisi od prirode samog zadatka, pa možemo razlikovati:

a) - zadatke koji su permanentno aktivni;

b) - zadatke koji se periodično izvode, uglavnom na poticaj sata stvarnog vremena;

c) - zadatke koji se izvode na osnovu zahtjeva za prekidom koje u slučajnim trenucima vremena postavlja vanjska okolina.

Treća skupina je ujedno i najmasovnija, a karakterizirana je time da se zahtjev mora ispuniti u stvarnom vremenu.

Rasporedjivanje zadataka bi zavisno o tipu zadatka trebalo vršiti u različitim vremenima:

- za zadatke navedene pod a) unaprijed - u fazi projektiranja sistema, tj. rasporedjivanje je statičko,
- za zadatke navedene pod b) i c) pogodnije bi bilo u fazi rada sistema, tj. rasporedjivanje je dinamičko.

Zadatak bi trebalo da bude pridijeljen izabranom računalu tako da se ostvari najbolji uticaj na ponašanje cjelokupnog sistema (minimalno vrijeme kašnjenja, minimalno vrijeme izvršavanja, maksimalna propusnost sistema itd.). Dinamičko rasporedjivanje bi trebalo uticati pozitivno na osobine sistema.

Prema funkcijama koje podržavaju zadatke možemo podijeliti na:

- zadatke koji obavljaju izlazno/ulazne transakcije i koji se izvode samo na odredjenim računalima u sistemu, te zbog periferije uz koju su vezani mogu biti rasporedjeni isključivo statički;

- zadatake koji manipuliraju nekim sredstvima i koji s se takodjer izvode samo na odredjenim računalima u sistemu, te zbog periferije uz koju su vezani mogu biti rasporedjeni isključivo statički;

- zadatke koji manipuliraju nekim sredstvima i koji se takodjer izvode samo na odredjenim računalima u sistemu, ali može postojati više zadataka koji obavljaju istovjetne funkcije (na pr. procesiranje teksta, aritmetika u tekućem zarezu etc.);

- zadatke koji vrše odredjena izračunavanja i koji se mogu izvoditi na bilo kojem od računala u sistemu.

Posljednje dvije grupe zadataka mogu biti dinamički rasporedjene.

Rasporedjivane ulazno/izlaznih zadataka se provodi u fazi kreiranja sistema (1) i dalje se u ovom radu neće razmatrati.

3. PRIJEDLOG RASPOREDJIVANJA ZADATAKA U SISTEMU

U literaturi se može naći više pristupa rasporedjivanju zadataka u višeprocesorskim i višeračunarskim sistemima. Svi oni su ekvivalentni u tom smislu da se sa logičkog stanovišta svaki od njih može upotrebiti za rješavanje proizvoljnog problema rasporedjivanja. Za dati problem, medjutim, neki od njih vode složenijim i neefikasnijim programima nego ostali. Dakle, sa praktičnog stanovišta oni očigledno nisu ekvivalentni.

Nijedan od autoru dostupnih algoritama za rasporedjivanje zadataka u sistemima sa više procesora nije eksplicitno uzeo u obzir dva veoma bitna faktora:

- medjuovisnosti zadataka koji se izvode na različitim mjestima u sistemu i koji komuniciraju izmjenom poruka da bi obavili postavljene zadaće;

- ovisnosti topologije mreže (prsten, zvijezda, linija itd.) na ukupno komunikacijsko opterećenje.

Ako se pridržavamo definicije raspodijeljenih sistema date u (2,3) ignoriranje medjuovisnosti zadataka koji se izvode na različitim mjestima u sistemu vodi do veoma loših osobina takvih sistema, pa algoritmi za rasporedjivanje zadataka koji ne uzimaju obzir medjuovisnosti zadataka nisu realistični, te njihova primjena neće poboljšati performanse sistema. Stoga algoritmi za rasporedjivanje zadataka koji ne uzimaju u obzir medjuovisnosti zadataka nisu realistični, te njihova primjena neće poboljšati performanse sistema.

Algoritmi za lokalno i globalno rasporedjivanje koji su definirani u (4,5) za razliku od sličnih postojećih algoritama uzimaju u obzir medjuzavisnosti zadataka koji se izvode na pojedinim računalima. Pošto su odluke o rasporedjivanju zadataka ovisne i o sklopovskoj gradji sistema, kao i o realizaciji programske podrške trebalo je prije nego se pristupi kreiranju algoritama za rasporedjivanje prethodno definirati model raspodijeljenog sistema.

Prvi zadatak pri tome je bio da se pojmu procesa da precizno značenje, tj. da se uvede nedvosmislena termi-

nclogija kojom se definira šta je proces i kako se on realizira u pojedinom računalu. Slijedeći korak je da se odaberu osnovne operacije za sinhronizaciju i prenos podataka izmedju medjuzavisnih procesa (6).

Na osnovu usvojenih komunikacijskih i sinhronizacijskih mehanizama bilo je moguće definirati algoritam za lokalno rasporedjivanje (5).

Pošto je komunikacija zsnovana na izmjeni poruka znatno opterećenje komunikacijske mreže može dovesti do veoma loših osobina sistema. Kao što je u (7) pokazano sa porastom nivoa problema istovremeno raste i potreba za obavještavanjem, te eventualnim učešćem ostalih dijelova sistema u njegovom rješavanju. Stoga je za definiranje algoritama za globalno rasporedjivanje od najvećeg značaja problematika izbora topologije komunikacijske mreže koja bi trebalo da obezbijedi pouzdan i brz prenos informacija medju računalima (7,8).

U (4) je predstavljen deterministički dinamički algoritam za rasporedjivanje zadataka u konkretnoj prstenastoj strukturi. Prstenasta topologija sistema izabrana je pod pretpostavkom da bi visoka iskoristivost veza i strukturna jednostavnost prenosa poruka mogla zadovoljiti odredjenu skupinu korisnika koji ne postavljaju prevelike zahtjeve na brzinu komuniciranja. Kod ovog metoda globalni rasporedjivač će na osnovu proračuna minimiziranja dodatnog komunikacijskog opterećenja (koje uvodi novi zadatak) iz skupa alternativa izabrati ono rješenje koje će u okviru zacrtanom algoritmom maksimalno zadovoljiti postavljene kriterije. Naime, medjuovisnost pojedinih zadataka je razlog da neki od njih ostaju duže u stanju čekanja. Pregled nad tom medjuovisnošću se dobija skupljanjem informacija o lokacijama komunikacijskih partnera i opterećenosti računala na kojima se oni nalaze. Nedostatak predloženog algoritma je u tome što sa porastom složenosti promatrane konfiguracije raste i dodatno neproduktivno vrijeme potrebno da se sakupe sve informacije.

Stoga je u nastavku definiran brži ali manje precizan algoritam za koji se neće tražiti optimalno već samo zadovoljavajuće rješenje problema.

3.1. Detalji realizacije

U ovdje predloženom algoritmu informacije o medjuovisnosti zadataka, tj. o lokaciji i imenima partnera zadatka dobijaju se na osnovu procjene (pogadjanja), uzimajući u obzir odredjene parametre koji indirektno pokazuju na medjuovisnosti zadataka. U nekim računalima zadaci čekaju duže na prijem poruke jer su računala na kojima se izvode njihovi partneri previše opterećena. Analiziraće se dva indikatora opterećenja - komunikacijsko i izvršno opterećenje. Ako pretpostavimo da računalo sadrži dosta zadataka u repovima rasporedjivač ne bi trebalo da rasporedi novi zadatak tom računalu mada kapacitet procesoru to dozvoljava. Promatraće se izvršno opterećenje uzimajući u obzir samo broj pripravnih zadataka. Algoritam za razliku od izloženog u (4) ne pretpostavlja da je vrijeme izvršenja svakog zadatka poznato.

Komunikacijsko opterećenje se definira kao količina komunikacijskih aktivnosti koje čekaju da budu obavljene. Što je komunikacijsko opterećenje veće duže će se čekati u repu komunikacijskog računala.

Dakle, predloženi algoritam pretpostavlja da se zadatak rasporedi računalu sa najmanjim brojem zadataka pripravnih za izvodjenje i najmanjim brojem poruka koje čekaju da budu prenesene.

Informacije potrebne za izvodjenje algoritma:

vk = f(ime) : vrijeme zakašnjavanja zadatka

n = F (ime,računalo) : označava da li je odredjeno računalo sposobno (zbog eventualnih sklopovskih ili programskih ograničenja) izvesti zadatak

D = 1 zadatak se može dodijeliti

D = 0 zadatak se ne može dodijeliti (računalo je posebne namjene)

b) Informacije koje će se dobiti od ostalih računala:

op_rač(i)=kom_opt(i)+izv_opt(i)

gdje je kom_opt(i) komunikacijsko a izv_opt(i) izvršno opterećenje računala "i".

Komunikaciono opterećenje kom_opt izraženo je sa:

kom_opt(i) = (PRED_POR)i + (PRED_POT)i

gdje je PRED_POR broj poruka, a PRED-POT broj potvrda koje bi računalo trebalo predati.

Izmjena poruka medju računalima zasnovana je na principu potvrde, te poruka može imati dva osnovna oblika koja bi se mogla predstaviti na slijedeći način:

```
TYPE poruka = RECORD
                  vrsta : (osnovna-poruka, potvrda-
                          prijema)
                  sadržaj: tekst-sadržaja
              END
```

Kako je u zaglavlju svake poruke sadržana adresa odredišta tačna vrijednost kašnjenja u svakom računalu može se izračunati primjenom procedure Kaš_kan (i,j) ne zahtijevajući pri tome nikakve dodatne informacije od ostalih računala:

```
Procedure Kaš_kan (i,j)

(* Ova procedura izračunava kašnjenje koje je uzrokovano *)
(* čekanjem u repu poruka dok se poruka prenosila izmedju*)
     (* računala "i" i računala "j",  *)
(* Ovo kašnjenje se računa zavisno o broju dionica *)
     (* koje je poruka trebalo da predje *)
(* Vrijeme potrebno da se poruka proslijedi izmedju sus-
   jednih *)
(* računala ovisi o komunikacijskom opterećenju pojedi-
   nog *)
```

```
(* računala vre_ček (i) koje se dobija putem *)
    (* procedure uzmi_nove podatke () : *)
IF (i   >  J) THEN
                BEGIN
                      s:=i; (* s je oznaka za računalo
                               sa manjim brojem*)
                      l:=j  (* a l je oznaka za računa-
                               lo sa većim brojem *)
                END
               ELSE
                BEGIN
                      s:=j;
                      l:=i
                END;
brd=br_di (i,j) (* izračunavanje broja dionica izmedju
računala "i" i "j", *)
(* Izračunavanje kašnjenja poruke u svim računalima na
   putu od "i" ka "j" *)
 kašnjenje :=0 ;
 IF (brd = ABS (i-j))
   THEN
     BEGIN
        (* izračunavanje kašnjenja u jednom smjeru *)
        FOR I = s TO (l-1) DO
           kašnjenje: = kašnjenje + vre_ček (i);
     END
   ELSE (* izračunavanje kašnjenja u suprotnom smijeru *)
     BEGIN
          FOR I = 1 TO n DO kašnjenje :=kašnjenje +
                vre_ček (i);
     END;
     BEGIN
          FOR I = 1 TO (s-1) DO kašnjenje : = kašnje-
                nje + vre_ček (i);
          END;
     RETURN (kašnjenje);
    END;
```

Izvršno opterećenje izv_opt (i) može se izraziti sa

$$izv\_opt\ (i) = \frac{R}{\rho_n}$$

T = A - W

W = Wpre+Wpri

R - broj pripravnih zadataka
A - broj aktivnih zadataka
W - broj zadataka koji čekaju na prijem (Wpre) ili predaju poruke (Wpri)

ρ_n je srednje vrijeme opsluživanja zadatka ovisno o broju instrukcija i o T_n gdje je T_n srednje vrijeme izvršenja pojedine instrukcije. T_n zavisi od instrukcionog repertoara i načina adresiranja konkretnog mikroračunala kao i od strukture samog programa. Za svrhe procjene se ovo vrijeme može definirati kao (9):

$$T_n = 2*Tadd$$

gdje je Tadd najkraće vrijeme potrebno da se izvrši instrukcija sabiranja sadržaja memorijske lokacije i akumulatora.

U nastavku je prikazan algoritam za rasporedjivanje:

```
Procedure rasporedjivač (ime);
(* Algoritam za odredjivanje koje računalo će preuzeti
   izvršenje novog zadatka *)
(* Ako je izvršenje zadatka nov_zad(ime) hit o ne čekaj
   na nove podatke, *)
(* iskoristi stare vrijednosti *)
IF NOT (urgent (ime)) THEN
   BEGIN
       uzmi_nove_podatke ():  (* sakupi opt_rač(i) od
                                 svih računala
       izabrano_računalo :=0; (* inicijalizacija počet-
                                 nih vrijednosti *)
       selekcioni_faktor :=max;
FOR i = TO (broj računala) DO
 BEGIN
  IF(D(ime,i)=1)then (* ispitivanje da li je računalo
     sposobno da preuzme zadatak *)
    BEGIN
      IF (opt_rač(i)  <selekcioni_faktor) THEN
            BEGIN
               izabrano_računalo :=i ;
               selekcioni_faktor i=opt:rač(i);
            END;
    END;
 END;
 IF NOT (izabrano_računalo = 0)
    THEN
       BEGIN
          START (ime, izabrano_računalo);
          RETURN (ime, izabrano_računalo);
          (* rasporedjivanje je uspješno završeno *)
       END;
    ELSE RETURN (ime,0) (* rasporedjivanje zadatka nije
                           obavljeno *)
 END, (* kraj procedure rasporedjivača *)
```

Prednosti ovog algoritma u odnosu na algoritam definiran u (4) su:

- identifikator zadatka sadrži malo informacija koje definira sam programer;

- faktori izvršnog i komunikacijskog opterećenja ovise o dužini repova unutar radnog i komunikacijskog računala. Ažuriranje stanja računala jezgro obavlja jednostavno i brzo svaki put kada je pozvano i dodatno neproduktivno vrijeme je veoma maleno;

- ciklične poruke koje sadrže informacije o stanju ostalih računala u sistemu su minimalne dužine (samo jedna vrijednost po računalu). Stoga one ne predstavljaju ozbiljno opterećenje komunikacijskog sistema, te mogu biti obradjene sa najvišim prioritetom.

Nedostaci su u tome što algoritam ne uzima u obzir lokacije na kojima se nalaze medjuovisni zadaci sa kojima se komunicira, nego donosi odluku samo na osnovu procjene opterećenja pojedinih računala.

Ocjena predloženog metoda data je na osnovu analitičkog postupka, a ovdje će se zbog ograničenog prostora samo naznačiti. Primjenom metoda masovnog posluživanja opisuje se općeniti proces posluživanja preko funkcija ulazaka, posluživanja i čekanja.

Pri ocjeni modela polazi se od slijedećih pretpostavki:

- dolazno vrijeme je raspodijeljeno po Poissonovom zakonu;
- zadaci se izvode po pravilu "prvi dolazi - prvi poslužen";
- vrijeme posluživanja ima eksponencijalnu razdiobu.

Na osnovu ovih pretpostavki primijenjen je modificirani M/g/m model koji je općenito karakteriziran Poissonovim procesom ulazaka sa srednjom vrijednošću i sa raspodjelom vremena posluživanja oblika H(Es), koja ima s srednju vrijednost vremena posluživanja E(s) i k-ti moment E(s).

Uz pretpostavke da su vremena iniciranja zadataka Poissonov proces i vrijeme usluživanja zadataka eksponencijalno rasporedjeno, pretpostavlja se da je intenzitet ulaznog toka zadataka takav da su sva računala opterećena, tj. promatran je sistem pod jakim opterećenjem.

Teorijska osnova za ovakav pristup je data u radovima (10-17). Procjena minimalnog broja računala koja bi mogla zadovoljiti postavljene zahtjeve data je u radu (15). Srednja dužina repa i srednje vrijeme boravka u sistemu izračunati su na osnovu Pollaczek-Khinchinove relacije date u radu (16). U (17) je dat program i predstavljeni rezultati simulacije na računalu VAX 780/11 koji se podudaraju sa ovako dobijenim vrijednostima.

5. ZAKLJUČAK

Problematika primjene raspodijeljenih sistema u vodjenju složenih procesa je od velikog teorijskog i praktičnog značaja. Za optimizaciju rada ovakvih sistema od najveće važnosti je pristup rasporedjivanju zadataka po računalima. U radu je izložena osnovna ideja i način razmišljanja vezan uz taj problem sa ciljem da se ukaže na nedostatke postojećih rješenja i poboljšanja koja bi se mogla uvesti.

Suštinska razlika u odnosu na druge, autoru poznate pristupe je u tome što se predlaže da u kriznim situacijama, kada je opterećenje sistema najveće izvodjenje zadatka koji vrši odredjena izračunavanja preuzme ono računalo koje je sa sistemskog stanovišta u tom trenutku najpodobnije. Time se (po principu spojenih posuda) ukupno opterećenje rasporedjuje jednoliko na sve učesnike, a jednoliko opterećen sistem je i sistem maksimalne pouzdanosti i raspoloživosti. Sistem je deterministički - kreiran je za unaprijed definirane ciljeve, ali je i dinamički, jer se sve do inicijalizacije zadatka ostavlja otvorenim pitanje koje od računala će ga izvesti.

Globalni rasporedjivač se sastoji od skupa identičnih algoritama koje izvode komunikacijska računala. Algoritam donosi odluku o izboru najpogodnijeg računala na osnovu proračunavanja izvršnog opterećenje i vremena čekanja u repu svakog računala. Informacije o medjuovisnosti zadataka, tj. o lokacijama komunikacijskih partnera dobijaju se na osnovu procjene (pogadjanja), uzimanjem u obzir odredjenih parametara koji indirektno ukazuju na medjuovisnosti zadataka.

Težilo se ka tome da predloženi algoritam sadrži odredjeni stupanj općenitosti, tako da je moguća njegova primjena (sa odgovarajućim malim izmjenama) i na druge tipove komunikacijskih i sinhronizacijskih mehanizama, kao i na druge arhitekture raspodijeljenih sistema.

LITERATURA:

1. Stone, H.: "Multiprocesor scheduling with the aid of network flow algorithms", IEEE trans. on soft, eng. vol. SE-3 no, 1 (Jan.1977).
2. Le Lann, G.: "An analysis of different approaches to distributed computing", Proc. of the ist intern.conf. of distributed comp.systems, Huntsville, AL, (oct. 1979).
3. Enslow, P.: "What is a distributed data processing system?", Computer, vol.11, no.1 (Jan.1978).
4. Kukrika,M.: "Pristup dinamičkom rasporedjivanju zadataka u prstenastoj mreži računala", Informatica 2 (1984).
5. Kukrika,M.: "Pristup kreiranju rasporedjivača zadataka u distribuiranom izvršnom sistemu sa radom u stvarnom vremenu", Informatica 3 (1984).
6. Kukrika,M.: "Primjer sinhronizacije medjuzavisnih zadataka u raspodijeljenim sistemima", XXVIII Jugoslovenska konferencija ETAN, Split (1984).
7. Kukrika,M.: "Problemi komuniciranja u sistemima sa više mikroračunala" II Jugoslovensko savjetovanje o mikroračunalima u procesnom upravljanju - MIPRO, (1983).
8. Kukrika,M.: "Pristup organiziranju lokalnih mreža mikroračunala" V medjunarodni simpozij "Kompjuter na sveučilištu", Cavtat (1983).
9. Kukrika,M., Štrkić G.: "Primjer višeprocesorskog sistema u vodjenju procesa u realnom vremenu", XXVII Jugoslovenska konferencija ETAN, Struga (1983).
10. Pauše, Z.: "Vjerojatnost, informacija, stohastički procesi", Školska knjiga, Zagreb (1980).
11. Basket,T.: et al. "Open,closed and mixed networks of gueves with different clasess of costumers", JACM Vol.22 no.2, (april 1975).

12. Chandy,K. and Sauer,C.: "Aproximate methods for analysing gueuing network models of computing systems", Computer sciences dept., University of Texas, Austin (1978).

13. Kleinrock, L.: "Que ing systems",vol.2: "Computer application, John Willey and Sons inc., New York (1976).

14. Kleinrock, L.: "Que ing systems", vol.1: Theory, John W Willey and Sons inc., New York (1974).

15. Beizer,B.: "Microanalysis of computer systems perfo-rmance", Van Nostrand Reinhold Company (1978) ACM Computing Surveys, vol.10, no.3 (sept. 1978).

16. Cofman, E and Denning, P.: "Theory of operating systems", Prentice Hall (1975).

17. Hadžiomerović,D.: "Primjena metoda masovnog poslu-živanja u višeprocesorskim sistemima", Diplomski rad, ETF Banjaluka (1983).

informatica 85

Vabilo k sodelovanju **Call for Papers**

Symposium and Seminars Informatica '85
Nova Gorica, September 24th–27th, 1985

Conference
18th Yugoslav International Conference on Computer Technology and Usage

Seminars
Selected Topics in Computer Technology and Usage
Nova Gorica, September 24th–27th, 1985

Exhibition
Exhibition of Computer Technology, Usage, Literature and Other Computer Equipment with International Participation
Nova Gorica, September 24th–27th, 1985

Deadlines

April 1, 1985 Submission of the application form and 2 copies of the extended summary

July 15, 1985 Submission of the full text of contribution

PAPER/SHORT PAPER/ TECHNICAL REPORT REGISTRATION

This application should be typewritten

1. Title of the Paper: .....
2. Extended summary (approximately 1000 words) should be enclosed.
3. Program Category of the Paper (circle appropriate choice)
   1. Survey of Technology and/or Usage
   2. System Architecture
   3. Process Control
   4. Systems Development Tools
   5. Small Business Systems
   6. Usage in Education
   7. Personal Computers
   8. CAD/CAM Systems
   9. Artificial Intelligence and Robots
   10. Computer Networks
   11. Fifth Computer Generation
4. Classification of the Paper (circle appropriate choice)
   1. Paper
   2. Short paper
   3. Technical report

5. Author(s): .....
   Organization: .....
   Street: .....
   Postal Code: ..... City: .....
   Country: .....
   Date: ..... Signature: .....

This application form together with two copies of extended summary must reach the following address before April 1, 1985: Informatica '85, 61116 Ljubljana, p.p. 2, Yugoslavia

PRIJAVA REFERATA / KRATKEGA REFERATA / STROKOVNEGA POROČILA

Prijavo izpolnite s pisalnim strojem

1. Naslov referata .....
2. Razširjeni povzetek (približno 1000 besed) priložite prijavi.
3. Programsko področje referata (obkrožite ustrezno številko)
   1. Pregled tehnologije in uporabe
   2. Arhitektura in zgradba računalniških sistemov
   3. Upravljanje procesov
   4. Sistemski razvojni pripomočki
   5. Mali poslovni sistemi
   6. Uporaba pri izobraževanju
   7. Osebni računalniki
   8. CAD/CAM mikrosistemi
   9. Umetna inteligenca in roboti
   10. Računalniške mreže
   11. Peta računalniška generacija
4. Razvrstitev referata (obkrožite)
   1. referat (pomembnejše delo)
   2. kratek referat
   3. poročilo

5. Avtorji: .....
   Delovna organizacija: .....
   Ulica: .....
   Poštna številka: ..... Kraj: .....
   Država: .....
   Datum: ..... Podpis: .....

Prijavnica, skupaj z dvema kopijama razširjenega povzetka, mora prispeti najkasneje do 1. aprila 1985 na naslov: Informatica '85, 61116 Ljubljana, p.p. 2

NOVE RAČUNALNIŠKE GENERACIJE

UVOD K NOVI RUBRIKI ČASOPISA INFORMATIKA

S to številko časopisa Informatica se odpira nova rubrika z naslovom

Nove računalniške generacije

(The New Computer Generations). Pridevnik nove je bil izbran zaradi dejstva, da se v tehnološkem jeziku jutrišnjih generacij že pojavljajo koncepti 5. in 6. računalniške generacije in še kasnejših generacij: kot da bo ta razvoj od določene stopnje naprej zelo hiter oziroma tehnološko intenziven. Tako se npr. že danes načrtujejo 256-bitni mikroprocesorji z novimi inmosovskimi (paralelnimi, celičnimi, riscovskimi) lastnostmi.

Širom po svetu potekajo v raznih središčih računalniškega znanja in izkušenosti raziskovalni in razvojni projekti, katerih cilj je odkrivanje zmogljivosti novih računalniških generacij. Potrebne so nove generacije procesorjev, ki naj bi odpravile omejitve sekvenčno delujočih računalnikov. Potrebni so novi jeziki, ki bi omogočili delo z znanjem in ne samo podatkovno manipulacijo pri dobivanju informacije. Potrebni so novi stroji, ki bi bili sposobni reagirati v obdajajočem svetu in ne samo delovati tako, kot jim je naročeno s programom. Izmišljene in razpoznane so bile nove generacije aparaturne, programske in robotske opreme, ki so izrasle in so bile gojene širom po svetu.

Prehod iz današnje 4. računalniške v 5. generacijo najbrž ne bo samo evolutiven, kot je bil prehod in 3. v 4. generacijo. Da bi lahko bila 5. računalniška generacija revolucionarna (v pozitivnem tehnološkem pomenu), morajo biti doseženi nekateri temeljni zmogljivostni cilji tkim. prvega tehnološkega vala, in sicer tako, da se uvedejo

-- nove materialne_proizvodne tehnologije z 100- do 1000-kratno gostoto današnje visoke integracije in s prav takšnimi povečanji signalnih in operacijskih hitrosti;

-- nove interaktivne računalniške in informacijske tehnike in metodologije za načrtovanje in proizvodnjo nove tehnologije, seveda s povečano hitrostjo in stopnjo kompleksnosti (pospešeni in napredni CAD_CAM sistemi, ki so podprti z opremo nove generacije);

-- novi sistemski in aplikativni programi, ki bodo po stopnji zapletenosti prekašali današnje največje programske sisteme za faktor 100 do 1000;

-- nove računalniške metode, tehnike in aplikacije za razvoj tako zapletene nove programske opreme;

-- novi principi na področju p a r a l e l n o s t i računalniške arhitekture, računalniškega programiranja in paralelne podatkovne organizacije;

-- nove periferne naprave.

Uvedba naštetih novih tehnik in tehnologij naj bi omogočila razvoj dovolj zapletenih in obsežnih programskih paketov v okviru novih generacijskih valov, kot so vobče

-- uporaba umetne in naravne (interaktivne) inteligence,

-- izvedeniški (ekspertni) sistemi in

-- pisna, zvočna in vidna uporaba naravnih jezikov in slikovnih predstav.

To naj bi bil drugi tehnološki val, ki bi pogojeval zares kompleksne, specializirane aplikacije na različnih področjih dela (specializirane baze z mehanizmi učenja, eksperte, CAD_CAM, robotiko), ki bi predstavljale začetek tretjega tehnološkega vala naslednjih generacij.

V naslednjem prispevku v tej rubriki so navedeni podatki o informacijah, ki so izšle v časopisu Informatica in se nanašajo na naslednjo računalniško generacijo oziroma na področja, ki so z njo povezana. V njih bo bralec našel precej materiala, ki je uporaben pri graditvi slike o prihodnji računalniški generaciji in njeni uporabi.

V prihodnje bomo v tej rubriki objavljali tudi obveščevalne prispevke s področij

osnovne metodologije umetne inteligence,

logičnega programiranja,

učenja sistemov (avtomatičnega in interaktivnega),

generiranja programirnih in programskih sistemov,

arhitekture prihodnjih računalniških generacij in njihove sistemske programske opreme,

izvedeniških sistemov,

paralelizma (programskega, arhitekturnega, funkcionalnega, konceptualnega),

razumevanja in generiranja naravnih jezikov, slušnih in vidnih (slikovnih) pojavov.

Zanimali nas bodo tudi komercialni izdelki prihodnjih generacij (aplikacije, programi, specializirani računalniški sistemi).

Rubrika Prihodnje računalniške generacije naj bi postala stalen obveščevalni projekt (infor-

macijski, podatkovni, kritični) časopisa Informatica, ki naj bi sistematično vključeval prispevke naših (strokovnih) dopisnikov iz Japonske, ZDA in Zapadne Evrope.

A. P. Železnikar

Doslej objavljene novice in prispevki s področja novih računalniških generacij v časopisu Informatica

(1) Jezik Lisp za mikroračunalnike. NIZ. Informatica 9 (1985), št.1, str.79.

(2) Peta računalniška generacija prihaja (intervju, B. Čerin). NIZ. Infotmatica 8 (1984), št.4, str.95-96.

(3) Jeziki in prevajalniki za umetno inteligenco. NIZ. Informatica 8 (1984), št.3, str.75-79.

(4) B.Džonova-Jerman, J.Žerovnik: Uporaba programskih grafov pri ugotavljanju vzporednosti v računalniških algoritmih II. Informatica 8 (1984), št.3, str.53-56.

(5) B.Džonova-Jerman, J.Žerovnik: Uporaba programskih grafov pri ugotavljanju vzporednosti v računalniških algoritmih I. Informatica 8 (1984), št.2, str.20-23.

(6) Nekatere nove knjige (W.Lowen: Dichtomies of the Mind). NIZ. Informatica 8 (1984), št.1, str.82.

(7) I.Stojmenović, V.Stojković, L.Jerinić, J. Mirčevski: O implementaciji prevodioca Lispkit Lisp-jezika na jezik SECD-mašine izvršenoj na Fortran-jeziku. Informatica 8 (1984), št.1, str.57-64.

(8) J.V.Knop, K.Szymanski, N.Trinajstić: Future Developments in Computer Architecture. Informatica 8 (1984), št.1, str.48-56.

(9) Evropska skupina za 5. generacijo. NIZ. Informatica 7 (1983), št.4, str.80.

(10) Problemi inteligence. NIZ. Informatica 7 (1983), št.1, str.79-80.

(11) Izvedenski sistemi so specializirani in ne vedo vsega. NIZ. Informatica 6 (1982), št. 4, str.81-83.

(12) Zapad mora ostati previden. NIZ. Informatica 6 (1982), št.4, str.80.

(13) Non-Von, paralelni mikroprocesorski sistem. NIZ. Informatica 6 (1982), št.4, str.79.

(14) Perspektive umetne inteligence. NIZ. Informatica 6 (1982), št.3, str.68-69.

(15) Japonski načrt računalniške dominacije. NIZ. Informatica 6 (1982), št.3, str.66-67.

(16) D.Bojadžijev, N.Lavrač, I.Mozetič: Izkušnja s Prologom kot jezikom za specifikacijo informacijskih sistemov. Informatica 6 (1982), št.3, str.54-58.

(17) I.Bruha: On an Implementation of the Pop-2 Language. Informatica 6 (1982), št.3, str.46-53.

(18) I.Bruha: Some Problems of Image Processing by Parallel Procesor Clip. Informatica 6 (1982), št.3, str.37-41.

(19) I.Bratko, I.Kononenko, I.Mozetič: An Efficient Implementation of Advice Language 2. Informatica 6 (1982), št.2, str.21-26.

(20) A.P.Železnikar: Informatizacija kot svetovni izziv. Informatica 6 (1982), št.2, str.13-15.

(21) P.M.Lavorel: Steps towards Natural Computation. Informatica 6 (1982), št.2, str.3-12.

(22) A.P.Železnikar: Informatizacija in tretji val. Informatica 6 (1982), št.1, str.3.

A.P.Železnikar

Izvedeniški sistemi, ki temeljijo na znanju I

1. Uvod

Izvedeniški (ekspertni) sistemi, osnovani na znanju, ali kratko s i s t e m i z n a n j a , uporabljajo človekovo znanje (poznavanje, vednost, spoznanje, izkustvo, veščino, spretnost) pri reševanju tistih problemov, za rešitev katerih bi bila sicer praviloma potrebna človekova inteligenca (razumnost, bistroumnost, pamet, razumevanje, znanje, poznavanje, izobraženost, učenost). Sistemi znanja predstavljajo in uporabljajo znanje r a č u n a l n i š k o (elektronsko, z računalniki, z računalniško metodologijo). Ta lastnost in oblika sistemov znanja naj bi iz njih naredila močnejša orodja, kot so bile prejšnje tehnologije shranjevanja in prenosa znanja, kot so knjige in navadni programi. Vse prejšnje tehnologije za shranjevanje in prenos znanja so obremenjene z bistvenimi omejitvami. Čeprav je v knjigah nakopičena največja količina znanja, je to znanje v bistvu s i m b o l i č n o in nedejavno (pasivno, ravnodušno, nepodjetno). Pri uporabi znanja iz knjig mora biti to znanje razpoznano, razloženo, utemeljeno (interpretirano, pojasnjeno, prikazano) in obstajati mora

odločitev, v kolikšni meri in obliki bo uporabljeno pri reševanju problemov.

Večina računalnikov izvaja opravila skladno z odločitveno logiko navadnih programov in ti programi niso brez nadaljnega prilagojeni bistvenim količinam znanja. Program je sestavljen iz dveh zvrsti: algoritmov in podatkov. Algoritmi določajo reševanje posebnih vrst problemov, podatki pa predstavljajo parametre posebnega problema. Človekovo znanje ni podobno takemu modelu, saj je sestavljeno iz osnovnih drobcev (fragmentov, odlomkov) izkustva (izkušenosti, sposobnosti, znanja) in uporablja nove načine organizacije drobcev pri odločanju o koristnosti novih struktur znanja.

Sistemi znanja zbirajo drobce znanja v bazah znanja in dostopajo v te baze skladno s posebnostmi problemov. Posledica tega je, da se sistemi znanja organizacijsko razlikujejo od navadnih programov, razlika pa je tudi v načinu, kako znanje vsebujejo, vključujejo, kako se izvajajo in kakšen vtis, vpliv oblikujejo s svojimi interakcijami.

Sistemi znanja simulirajo (hlinijo, ponarejajo, posnemajo) človekove izvedeniške zmogljivosti in predstavljajo za uporabnika človeku podobno ospredje (fasado, vnanjost, videz, občutje). Nekatere sodobne uporabe metod znanja obsegajo:

-- medicinsko diagnostiko,
-- popravljanje (reparacijo) naprav,
-- sestavljanje (konfiguriranje) računalnikov,
-- predstavljanje kemičnih podatkov in strukturno pojasnjevanje,
-- razumevanje govora in slik,
-- finančno odločanje,
-- signalno interpretiranje (razlaganje),
-- raziskave mineralov,
-- vojno obveščanje in planiranje,
-- svetovanje uporabe računalniških sistemov in
-- načrtovanje vezij z visoko integracijo (VLSI).

Na teh, naštetih področjih so sistemski razvijalci prepletli splošne metode znanja s posebnimi izkustvi danega problemskega (uporabniškega) prostora. Na naštetih področjih so se potrebe po metodah znanja pojavile zaradi omejitev alternativnih metod. Razvijalci so želeli vključiti v te sisteme veliko količino fragmentarnega (razdrobljenega), preudarnega (uvidevnega, razumevajočega, razsodnega) in izkustvenega znanja; želeli so reševati avtomatično tiste probleme, ki potrebujejo stroj za različne možnosti obnašanja, ki so glede na razpoložljive podatke najbolj ustrezne; želeli so sisteme, ki lahko uporabljajo svoje znanje za pojasnjevanje obnašanja na zahtevo.

2. Umetna inteligenca in tehnika znanja

Tehnika znanja (knowledge engineering) (kratko TZ) je posebna veja umetne inteligence (UI), ki uporablja znanje za reševanje problemov, za katero bi sicer bila potrebna človekova pamet. Tri zamisli (koncepti) UI obsegajo dejavnost TZ, in sicer:

-- simbolično programiranje,
-- reševanje problemov in
-- iskanje.

S i m b o l i č n i p r o g r a m i obdelujejo simbole, ki predstavljajo objekte in relacije. Ti programi uporabljajo izraze s spremenljivkami, te pa označujajo razrede objektov ali relacij med razredi objektov. Simbolični programi imajo še pogojne izraze, ki označujejo možne vrednostne prireditve spremenljivkam in uporabljajo pravila sklepanja ali deduktivne vzorce, ko določajo, katere vrednosti spremenljivk se izvajajo (deducirajo) iz drugih vrednosti. Simbolni programi lahko tako obdelujejo modele bitnosti realnega sveta, sisteme, ki v njih sodelujejo in učinke dejanskih ali navideznih sprememb v teh sistemih. Simbolični programi imajo navadno obliko seznamov (list) simboličnih izrazov. Ko se ti programi izvajajo ali interpretirajo, določajo vrednosti ali proizvajajo obnašanja kot stranske učinke izračunov. Ti programi so simbolični podatki za druge programe, ki na njih delujejo, jih modificirajo oziroma obdelujejo. Simbolični program ima hkrati dva obraza: je p r o g r a m in p o d a t e k .

Programi za reševanje problemov uporabljajo negotove in nedoločne metode za doseganje ciljev. Cilji so izjave (propozicije) o potrebnih značilnostih nekega prihodnjega sveta. Programi za reševanje problemov morajo poiskati primerno pot, ki vodi do potrebne prihodnje situacije. Sistem za razumevanje govora poskuša npr. zaznati (opaziti, spoznati, razlikovati) pomen v izgovoru (izrekanju, izražanju, načinu izražanja, govoru); njegov cilj je (gledano abstraktno) v identifikaciji (zaznavanju) besednega zaporedja, ki bi lahko produciralo sprejeto zvočno energijo in v primerni interpretaciji govotnikovih namer. Sistem za načrtovanje poskusov pa poskuša npr. določati zaporedje akcij, ki bodo po izvršitvi proizvedle potreben poskusni rezultat.

Sistemi za reševanje problemov morajo vobče iskati rešitve. Ko naletijo na problem, ne morejo navadno neposredno pristopiti k njegovemu reševanju in posredovati odgovor. Lahko uporabljajo vrsto iskalnih metod in lahko imajo verjetnostni reševalni generator, ki jim omogoča preštevanje vseh mogočih rešitev. Lahko generirajo in preizkusijo vse možnosti, dokler ne najdejo rešitve. Množica možnosti se imenuje i s k a l n i p r o s t o r , sama iskalna metoda pa n e u m n a (surova, brezčutna) s i l a . Veliko navadnih problemov, kot sta npr. popravilo vezij ali igranje iger, ima iskalni prostor astronomskega obsega. Problemski reševalnik lahko uporabi tudi vrsto palčnih (prstnih) pravil za izboljšanje svoje iskalne zmogljivosti. Ta pravila lahko podpirajo obetajoče kandidate, odvračajo kandidatske rešitve z majhno možnostjo uspeha, predlagajo enostavne preizkuse za zgodnje izločanje kandidatov iz obravnave ali označujejo lastnosti tistih trenutnih problemskih podatkov, ki nakazujejo obe-

tajoče zasledovalne smeri. Ta hevristična palčna pravila omogočajo problemskim reševalnikom obvladovanje kompleksnosti (zapletenosti) težavnih problemov.

3. Zgodovina umetne inteligence (UI) in tehnike znanja (TZ)

Umetna inteligenca (UI) je nastala v 50-ih letih, ko so računalniški raziskovalci pri reševanju problemov začeli uporabljati računalnike za pisanje simbolnih programov. Dobri rezultati v avtomatični dedukciji in pri reševanju problemov so povzročili dokajšnjo vznemirjenost in optimizem. Ko je program, ki so ga imenovali logični teoretik, dokazal skupino izrekov iz Principia matematica z uporabo hevurističnih metod reševanja problemov, se je večini zdelo razumljivo, da bodo večji in hitrejši računalniki lahko razširili moč splošnih problemskih reševalnikov na vse izzivajoča področja duhovnih dejavnosti.

V zadnjih dveh desetletjih so se raziskovalci UI naučili ceniti vrednost reševanja problemov na področjih značilnih znanj. Večina duhovnih problemov se ne rešuje z uporabo splošne strategije reševanja problemov. Za reševanje problemov na področjih človekove izkušenosti, kot so npr. tehnika, medicina ali programiranje, morajo strojni reševalniki znati to, kar znajo človeški reševalniki v povezavi z določenimi subjekti. Čeprav ima računalnik vrsto prednosti pred človekom, kot sta hitrost in konsistentnost (stalnost, trajnost, obstojnost), pa ne more nadomestiti svoje nevednosti in omejenosti. Raziskovalci UI so spoznali, da visok IQ (inteligenčni kvocijent) še ne naredi človeka za izvedenca; to se zgodi šele pri posebni izkušenosti. Da bi hiter in konsistenten simbolni procesor deloval kot človeški izvedenec, mu mora biti posredovano izkustvo, ki je primerljivo z izkustvom človeškega izvedenca. V tej smeri lahko napreduje tudi tehnika znanja.

TZ se je najprej razvila na univerzah in je pospešila prilagajanje zmogljivostim človeških izvedencev. Dendral ((3)) in Macsyma ((4)) sta najprej dosegla izvedeniško zmogljivost. Dendral ugotavlja molekularno strukturo snovi iz njenih masnospektrografskih in magnetnorezonančnih podatkov. Macsyma obdeluje in poenostavlja zapletene matematične izraze. Oba sistema sta se pojavila pred več kot desetimi leti in sta s svojo specializacijo prekosila njune človeške kreatorje in vse druge človeške strokovnjake.

V začetku 70-ih let se je pojavilo več aplikacij TZ. Proti koncu desetletja so nastali pomembni dosežki:

-- Mycin vsebuje 400 izkustvenih pravil, napisanih v angleškem IF-THEN formalizmu in se uporablja za diagnozo in ugotavljanje nalezljivih krvnih bolezni; njegova glavna privlačnost je, da lahko nejasno razloži vsak sklep ali vprašanje, ki ga je generiral.

-- Hearsay-II je večkratni, neodvisni in sodelujoči izvedeniški sistem, ki komunicira z uporabo globalne podatkovne baze (blackboard), da bi razumel sprejeti govor.

-- R1 ima približno 1000 pravil tipa IF-THEN in se uporablja za konfiguriranje naročil računalniških sistemov VAX; s tem odpravlja potrebe po najemanju in šolanju kadrov, ki bi to delo opravljali.

-- Internist ima približno 100000 izjav o relacijah med boleznimi in simptomi v interni medicini in ima znanje in sposobnost reševanja problemov, ki presega znanje večine specialistov v interni medicini.

4. Trenutno stanje UI

V začetku tega desetletja je bilo spoznano, da sta UI na splošno in TZ še posebej dozorela do tiste meje, da ju je mogoče uporabiti za komercialne in vladne (upravljavske) namene. To spoznanje je pospešilo razvoj področja in specializacijo v podpodročjih.

Tri glavna podpodročja UI dane so:

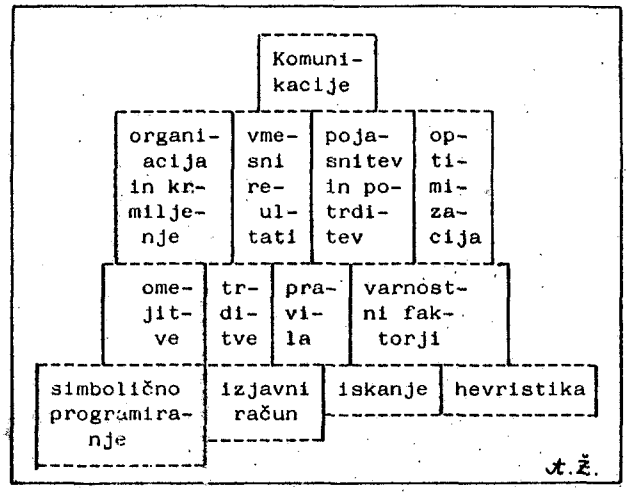
-- obdelava naravnih jezikov (ONJ),
-- videnje in robotika in
-- tehnika znanja (TZ).

Glavni projekti v ZDA se izvajajo pod okriljem DARPA, NIH in NSF. Japonska in Združeno kraljestvo sta oblikovali nove programe s podporo svojih vlad. Znatno komercialno zanimanje se je pojavilo na vseh treh podpodročjih, tako da obstajajo komercialne aplikacije tehnoloških in komercialnih orodij, ki podpirajo dodatne aplikacije. Tako prodaja IBM zgodnji produkt za ONJ podjetja AI Corporation pod imenom Intellect. Več proizvajalcev programske opreme je najavilo proizvode tipa ONJ za osebne računalnike. Več podjetij ponuje sisteme za videnje, izvedeniške sisteme in orodja za gradnjo lastnih izvedeniških sistemov. Podjetje Teknowledge gradi sisteme znanja po naročilu. Tako je najavilo izboljšave običajnih sistemov znanja za komercialne stranke, kot je npr. vrtalni nadzornik za naftne družbe, nadzornik računalniške programske opreme za uslužnostna podjetja, ki opravljajo podatkovno analizo, registracijo naročil in preizkus konfiguracij za elektronska podjetja in inteligentnega pomočnika za partnerje v velikih računovodskih podjetjih.

TZ obeta dolgo in obetavno življensko pot. Tehnologija TZ ima namen oblikovati pripomočke, ki bodo človeku omogočali zajemanje, shranjevanje, razdeljevanje in uporabo znanja z elektronskimi napravami. Zgodnje aplikacije UI nakazujejo njeno zmogljivost in ekonomsko pomembnost in kažejo na zelo široko področje možnosti njene uporabe. Vrsta zanimivih in pomembnih aplikacij pa presega zmogljivosti današnjih sistemov; ta deficit je tehnološki in seveda na področju znanja oziroma metodologije.

5. Metode, ki se uporabljajo v sistemih nanja

Slika 1 prikazuje gradnike sistema znanja. Osnovna ravnina je sestavljena iz tistih tehnik, ki so podlaga skoraj vsem aplikacijam. Te tehnike so: simbolično programiranje, izjavni račun, iskanje in hevristika. Izjavni račun doslej ni bil omenjen, vendar ga velja upoštevati.



Slika 1. Tehnike, ki se uporabljajo v sistemih znanja.

Sistemi znanja reššujejo probleme tako, da generirajo kandidatske rešitve in jih ocenjujejo. Pri reševanju se uporabljajo hevristična pravila na danih podatkih, ko se deducirajo logične ali verjetnostne posledice in ko se dokazuje, da' e posledice zadoščajo ciljem. Te akcije ustrezajo osnovnemu mehanizmu sklepanja in dokazovanja v izjavnem računu. Čeprav večina današnjih sistemov znanja dejansko ne uporablja programov formalne logike, se dosegajo z njimi enaki učinki. Izjavni račun razpolaga s svojim formalnim fundamentom, ki omogoča le omejeno sklepanje in dokazovanje.

Na drugi ravnini uporabljanih tehnik slike 1 imamo često načilne oblike predstavitve znanja: omejitve, trditve, pravila in varnostne faktorje. Primera omejitev sta: +Dva fizična predmeta ne moreta v istem času zasedati isti prostor+ in +Vsaka ugodnost v polici življenskega zavarovanja mora biti finančno utemeljena z zdravjem zavarovanca+. Sistemi znanja vključujejo omejitve za izražanje pridržkov o dopustnih stanjih, vrednostih in sklepih. Nekateri sistemi znanja izvajajo svoje vrednosti prvenstveno z razpoznavanjem in zadoščevanjem množicam zapletenih simboličnih omejitev. Na ta način razširja tehnika znanja (TZ) razred omejitvenih in zadostitvenih problemov na računalniško področje. Medtem ko so se računalniški sistemi osredotočili na linearne omejitve, pa se sistemi znanja sklicujejo predvsem na simbolične omejitve, kot so npr. zahteve po prostorskih, časovnih in logičnih relacijah.

Trditvene podatkovne baze razpolagajo s pripomočki za shranjevanje in razpoznavanje izjav. Trditev ustreza pravilni izjavi, faktu. Primeri trditev so npr. +Janez ima rad Micko+, +Sultan je pes+ in +Sosedovemu psu je ime Sultan+. Več enostavnih trditvenih oblik se zbere v relacijski podatkovni bazi, zapletene trditvene oblike pa ostanejo izven nje. Večina današnjih sistemov znanja vključuje svoje lastne, specializirane trditvene podsisteme podatkovnih baz.

Pravila predstavljajo deklarativno ali imperativno znanje posebne oblike. Poglejmo primer imperativnega pravila: +Če opazuješ bolnika z vročino in nahodom, potem lahko predpostavljaš, da ima bolnik gripo.+ To pravilo narekuje sistemu znanja, kako naj se obnaša. Relacijsko deklarativno pravilo pa lahko pove sistemu, kaj sme verjeti in kako je to pravilo nedoločeno: +Če ima bolnik gripo, potem bo tak bolnik verjetno imel vročino in nahod.+ Večina sistemov znanja uporablja eno od obeh oblik pravil. Vobče opisujejo deklarativna pravila način, kako delujejo stvari v svetu. Imperativna pravila pa predpisujejo hevristične metode, ki naj bi jih sistemi znanja uporabljali pri svojih lastnih operacijah.

Varnostni faktorji označujejo ravnino zaupanja ali veljavnosti, ki naj bi jo imel sistem znanja glede na svoje podatke, pravila in sklepe. Ti varnostni faktorji lahko odražajo raznovrstnost shem, ki ocenjujejo napake in nezanesljivost. Nekateri sistemi uporabljajo Bayesove pogojne verjetnosti za izračun varnosti (ustreznosti, gotovosti). Drugi uporabljajo v celoti subjektivne sisteme; npr. vrednost 1,0 povzroči gotovost, vrednost -1,0 povzroči gotovost izjavne negacije in vrednost 0,0 označuje pomanjkanje mnenja ali evidence. Čeprav se vlagajo veliki napori v izboljševanje tehnologije faktorjev varnosti, so lahko ti napori brezplodni. Sistemi znanja morajo natančno ocenjevati moč svojih sklepov, ker obstajajo neveljavne in formalne alternative; s količino formalizacije namreč ni moč izločati subjektivnih kakovosti odločitvenih procesov. Pa tudi več shem alternativnih varnostnih faktorjev lahko deluje enako zadovoljivo. Sistemi znanja delujejo zadovoljivo, ker lahko posnemajo človekove zmogljivosti. Človek ne bi reševal problemov zadovoljivo, če bi izračunaval zapletene matematične formule za določevanje svojih varnostnih faktorjev. Človek rešuje probleme zadovoljivo, ker deluje njegovo znanje dovolj dobro, je učinkovito, obstojno in kakovostno pri reševanju pomembnih problemov. Sistemi znanja naj bi enostavno uporabljali moč človekovega znanja.

Na tretji ravnini tehnik kaže slika 1 organizacijo in krmiljenje, vmesne rezultate, pojasnjevanje in potrjevanje in optimizacijo. Sistem znanja organizira in krmili svojo dejavnost skladno z arhitekturnimi oblikovalnimi principi, ki jih vsebuje. Npr. da bi našel zadostno potrditveno evidenco, lahko diagnostični medicinski izvedeniški sistem deluje nazaj od vseh njemu znanih potencialnih bolezni. Najprej lahko upošteva bolezen, ki je najbolj verjetna. Potem lahko zahteva evidenco o najbolj verjetnih in značilnih sindromih. Šele ko je

nabral preveliko količino protislovnih podatkov, lahko začne s preučevanjem naslednje možne bolezni. Izvedeniški sistem, ki deluje na ta način, uporablja t.i. globinsko, nazajšnjo verižno krmilno shemo. Vsaka posamezna krmilna shema lahko zahteva ustrezno organizacijo baze znanja in ustrezno prirejen mehanizem za sklepanje, ki išče in uporablja znanje. Tako sta krmiljenje in organizacija tesno povezana.

Vmesni rezultati se pojavljajo v vseh sistemih. Ker lahko sistemi znanja zajamejo bistvene zahteve zapletenih zmogljivosti, nastanejo potrebe za učinkovito uporabo vmesnih rezultatov. V nazajšnjem verižnem sistemu se npr. lahko pojavi zahteva o skupni evidenci več možnih alternativ, ki so v postopku preučevanja. Gradnja te evidence lahko zahteva veliko število računalniških operacij in sklepanja. Ko je sistem znanja ocenil to evidenco, mora shraniti rezultat, da ga bo lahko kasneje uporabil.

Ker sistemi znanja na splošno pojasnjujejo in potrjujejo svoje rezultate, so postali zanimivi za širok krog potencialnih uporabnikov. Končni uporabniki na različnih aplikativnih področjih potrebujejo priporočila sistemov znanja. Zaradi možnosti pojasnitve, kako je sistem znanja oblikoval svoje sklepe, dobi uporabnik vtis, da deluje sistem smotrno. Tako je mogoče pokazati, kako nastane iz množice predpostavk in zbirke hevrističnih pravil določen sklep. Za uporabnika je takšno pojasnjevanje pomembno prav tako kot pravila sama. Te pojasnjevalne lastnosti sistema znanja lahko seveda uporabijo tudi drugi subjekti, ki so v interakciji s sistemom. Vzdrževalci baze znanja, ki vključujejo v svoje delo izvedence in tehnike, tako kontinuirano izboljšujejo sisteme znanja in zvišujejo njihove zmogljivosti.

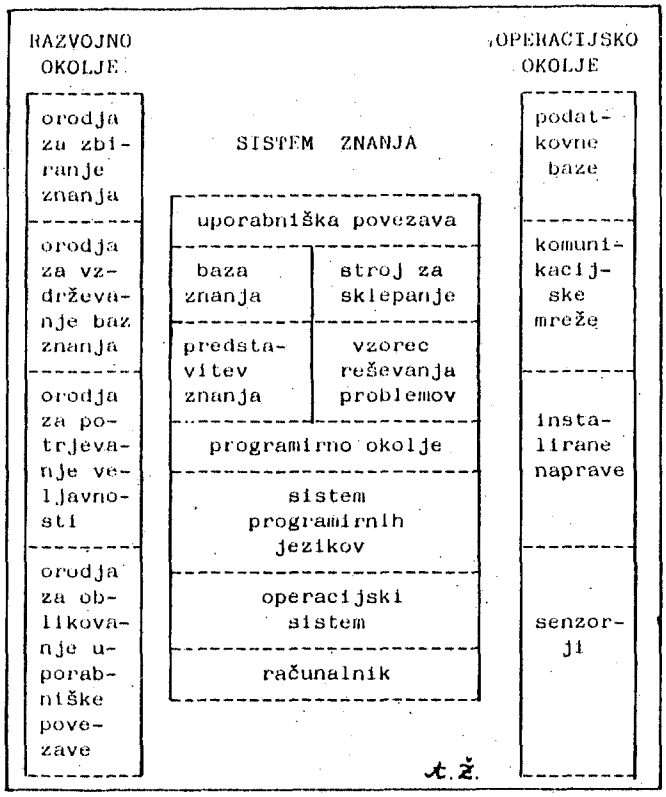
Optimnizacijske metode imajo pomembno vlogo v sistemih znanja. Sistemi znanja morajo tako kot drugi aplikativni računalniški sistemi opravljati svoje naloge tako hitro, kot je potrebno. Veliko današnjih aplikacij sistemov znanja deluje interaktivno z uporabniki tako pogosto, da mora čakati na zahtevane vhodne podatke. V teh primerih morajo tehniki znanja zagotoviti ekspertni dialog med sistemom in človekom. Dialog mora biti na določen način optimiziran. Nova orodja za gradnjo sistemov znanja uporabljajo izboljšane metode za specifikacijo imperativnega znanja in za njegovo učinkovito ločevanje od deskriptivnega znanja v dani problemski domeni. Najbolj pomembno območje optimizacije v zapletenih problemskih domenah je povezano z zmogljivostjo reševanja problemov danega sistema znanja: Ali ta sistem generira in preizkuša kandidatske rešitve v učinkovitem zaporedju? Ali se izogiba redudantnim izračunom? Ali prevaja simbolična pravila dovolj učinkovito? Ali razpoznava dovolj učinkovito trditve? Ali zna transformirati bazo znanja v ustreznejšo organizacijsko obliko za specialna opravila, ki uporabljajo učinkovitejše algoritme? Pri nekaterih aplikacijah je optimizacija sistemov znanja skrajšala izvajalne čase na tisočinko procenta prvotnega izvajalnega časa.

Temelj različnih tehnik v sistemih znanja so njihove komunikacijske zmogljivosti. Sistemi znanja namreč komunicirajo z izvedenci za sisteme znanja, s končnimi uporabniki, podatkovnimi bazami in drugimi računalniškimi sistemi. Tako kot človek dostopa in deluje z različnimi viri, mora tudi sistem znanja komunicirati z viri v lastnem, primernem jeziku. Sistemi znanja komunicirajo s tehniki znanja prek strukturnih urejevalnikov, prek katerih je možen dostop do določenih komponent in njihova modifikacija (npr. v bazi znanja). Sistemi znanja komunicirajo z izvedenci s posebnim dialogom, ki vsebuje pojasnitve, s katerimi se osvetljujejo reakcije sistema, tako da se izvedenci lahko odločajo za ustrezne spremembe v bazi znanja.

Pri komunikaciji s končnimi uporabniki lahko sistemi znanja uporabijo procese v naravnih jezikih, ko generirajo vprašanja ali odgovarjajo in interpretirajo uporabnikove odgovore. Nekateri današnji sistemi znanja uporabljajo video diske za razpoznavanje slik in za prikaz ukaznih zaporedij končnim uporabnikom. Razen te interakcije s človekom pa sodeluje sistem znanja praviloma tudi z drugimi računalniškimi sistemi.

Sistemi znanja morajo često formulirati in izvrševati navadne podatkovnoobdelovalne aplikacije, in sicer kot podopravila. Sistemi znanja so se razvili kot +novi možgani+, podobno kot pri višjih živalskih vrstah, kjer opravljajo funkcije nad +starimi možgani+. Pri specializiranih sistemih znanja je ta visoka funkcija predvidena za uporabo zapletenih računalniških programov, kot so npr. programi za strukturno tehniko in seizmično analizo. Sistemi znanja imajo pripomočke, s katerimi dostopajo k in razpoznavajo informacije iz sprotnih podatkovnih baz. Sistem znanja naj bi zmogel medsebojno zmesiti različne (tudi nasprotne) vire znanja iz različnih podatkovnih baz, z različnimi formati in kodiranjem in s produkcijo smiselne in integrirane interpretacije. Te potrebe se često pojavljajo v kompleksnih organizacijah, kot so npr. sprejemanje naročil in proizvodni sistemi v velikih podjetjih, obveščevalne in analizatorske funkcije v obrambnih sistemih itd.

Slika 2 prikazuje glavne komponente sodobnega sistema znanja in ga postavlja v kontekst njegovega okolja. Ta slika opisuje sistem znanja kot računalniško aplikacijo z razlikovanimi razvojnimi in opratívnimi okolji. Ljudje, ki so soudeleženi pri razvoju in razširitvi sistema znanja, uporabljajo orodja, ki so na sliki prikazana, in sicer orodja za zbiranje znanja, vzdrževanje baze znanja, za potrjevanje in za oblikovanje povezav. S temi orodji gradijo sisteme znanj, ki vsebujejo tri ključne komponente, in sicer: bazo znanja, stroj za sklepanje in uporabniški vmesnik (povezavo). Orodje vsebuje navadno tudi organizacijsko in krmilno metodo, ki določa specifični vzorec problemskega reševanja, ki ga bo sistem znanja uporabil. Ko je sistem znanja izpopolnil razvoj, preide v delovanje. V takem okolju navadno dostopa v podatkovne baze, povezuje različne komunikacijske mreže, se povezuje ali integrira z obstoječimi napravami in lahko sprejema podatke neposredno iz senzorskih sistemov.



Slika 2. Bistvene komponente sodobnega sistema znanja: z razvojnimi orodji na levi se pridobiva (priučuje, dosega) in vzdržuje znanje in to znanje se potem operativno uporablja.

Vrtalni nadzornik se npr. ukvarja s problemi obtičanja (zataknitve pri vrtanju) in pretegnitve (utrujenosti), ki se lahko pojavita v procesu vrtanja naftnih vrtin. V vrtini lahko sveder naleti na velike zaviralne (obtičalne) sile, ki so posledice trenja med geološkimi plastmi in vrtalnimi cevmi, stabilizatorji itd. Med vrtanjem mora sistem znanja dostopati v sprotno podatkovno bazo, kjer so shranjena poročila o vrtalnih operacijah, ki opisujejo ključne parametre. Sistem mora komunicirati z regionalnim ali centralnim operativnim upravljanjem, da bi dobival popravke in dopolnitve za bazo znanja in da bi lahko oddajal svoja poročila. Sistem mora delovati v surovem realnem okolju in je tako povezan s posebno, terensko opremo. Dostopati mora neposredno k senzorjem, ki oddajajo podatke o vrtanju v globinah in o pritisku v vrtalnen grezu.

Vrtalni nadzornik vsebuje npr. predstavitev znanja in paradigmo (vzorec) reševanja problemov orodij izvedeniškega sistema (npr. sistem KS300 podjetja Teknowledge). V bazo znanja se dostopa v primerih zataknitve in v njej je shranjenih približno 300 hevrističnih pravil in opisov za približno 50 ključnih vrtalnih parametrov. Stroj za sklepanje vodi dialog z uporabnikom v angleščini ali francoščini. Ta dialog posnema vsebinsko in sekvencialno način vpraševanja in analiziranja človeškega izvedenca, ki je bil uporabljen kot model. Vsaka hipoteza se preizkusi globinsko in z metodo nazajšnjega verižnega sklepanja, ki začenja z najbolj pogostnim zataknitvenim problemom in se nadaljuje z zbiranjem potrebne podporne evidence. Vsaka hipoteza, podatek ali hevristično pravilo lahko odraža negotovost. Sistem znanja kombinira te negotove količine in določa faktor varnosti (zanesljivosti) med vrednostma -1 in +1 za vsako potencialno diagnozo. Za vsako diagnozo, ki preseže vrednost varnostnega faktorja 0,2, oblikuje vrtalni nadzornik načrt rešitve trenutnega problema in minimizira verjetnost problemskega vračanja.

Vrtalni nadzornik deluje trenutno v programskem okolju, ki ga sestavljata KS300 in Interlisp, ki je osnovni programirni sistem; nadzornik se izvaja na računalnikih podjetij DEC in Xerox pod ustreznimi operacijskimi sistemi.

Razvojno okolje je setavljeno iz orodij, vsebovanih v KS300 in ta pomagajo tehnikom znanja in izvedencem, da doseže sistem izvedeniško znanje. Pri tem se uporablja angleški jezik in okrajšani zapis za prikazovanje in vnos pravil, za brskanje po bazi znanja, za strukturo, urejevanje, knjižnice primerov in za avtomatično preizkušanje. V tečaju za razvoj sistema so se tudi eksperti naučili uporabe orodij, tako da so lahko vplivali na kasnejše zbiranje znanja in na vzdrževanje baze znanja. Vsakokrat ko izvedenec ali vzdrževalec baze znanja spremeni pravilo, potrdi sistem KS300 avtomatično veljavnost sistema znanja, tako da opravi preizkus korektnosti za dani primer. Končno pa sistem znanja sodeluje z vrtalnim nadzornikom v naravnem jeziku, ko avtomatično prevaja, zbira in formatira ustrezne tekstovne fragmente, ki so povezani z elementi baze znanja. Povezovalna orodja omogočajo sistemu znanja, da proizvaja smiseln in pameten dialog z uporabo kratkih opisnih fraz, povezanih z vrtalnimi parametri, ki so bili izvedeniško predvideni. Vrtalni nadzornik tudi grafično prikazuje več ključnih dejavnikov, vključno z verjetnimi zataknitvenimi problemi, kamenitimi sloji in z razmerami na koncu vrtine. S standardnimi orodji paketa KS 300 se dinamično prikazujejo alternativne reakcije, vmesni sklepi in hevristična pravila, ki so trenutno v središču pozornosti.

6. Temelji tehnike znanja

Tudi TZ ima svojo teorijo in prakso. Tu lahko zasledimo tri glavne smeri. Sistemi znanja rešujejo probleme, ki bi sicer zahtevali človekovo pamet; to reševanje je lahko umetno ali naravno inteligentno. Organizacija in oblikovanje danega sistema znanja morata upoštevati tip in zapletenost problema, pa tudi moč in obliko izkustvenega znanja, ki je za reševanje na voljo; čeprav je življenska doba TZ kratka, se je vendarle razvila v smeri organiziranja sistemov znanja za različne primere. Znanje vsebuje sposobnost inteligentnega delovanja, toda običajno ne razpolaga s pripomočki za izkoriščanje in organizacijo tega potenciala; pri gradnji sistemov znanja mora tehnik znanja to znanje šele razviti in ga pretvoriti v uporabno obliko. Opišimo osnove takega razvoja.

6.1. Osnovne ideje

Ko govorimo o znanju v povezavi s tehniko znanja, mislimo na take podatke, ki bi lahko izboljšali učinkovitost problemskega reševalnika. Trije glavni tipi znanja uztrezajo temu opisu: fakti, ki izražajo veljavne izjave; prepričanja, ki izražajo verjetne izjave in hevristika (izkustvo), ki izraža pravila dobre presoje položajev, za katere vobče ne obstajajo zanesljivi algoritmi. Izvedenci se medseboj razlikujejo po kakovosti in količini znanja, ki ga imajo; vedo več in to, kar vedo, zvišuje njihovo učinkovitost.

Veliko človeških strokovnjakov opravlja naloge, ki zahtevajo veščinsko, samozavestno (trdilno) in informirano presojo. Te zahteve se pojavljajo zaradi zapletenosti, dvoumnosti ali nezanesljivosti razpoložljivih podatkov in problemsko reševalnih metod. V nasprotju z navadnimi aplikacijami obdelave podatkov deluje večina sistemov znanja v pogojih, ki ne zagotavljajo optimalnih ali pravilnih rešitev. V takih primerih mora problemski reševalnik uravnotežiti kakovost proizvedenega odgovora in napor, ki je pri tem potreben. Izvedenec najde navadno najboljši kompromis tako, da spregleda način, kako najti sprejemljiv odgovor s primerno porabo razpoložljivih virov.

S takšno pragmatično usmeritvijo zmogljivosti pridobijo pametni problemski reševalniki lastnost izboljšane učinkovitosti. Še posebej lahko izboljšave v hitrosti in izbiri producirajo sprejemljive rešitve bolj lahkotno in prispevajo k temu, da najde problemski reševalnik boljše rešitve v razpoložljivem času in reši še dodatne probleme. Na kakšen način lahko pameten problemski reševalnik (PPR) izboljša svojo učinkovitost?

(1) PPR poseduje znanje, ki se često uporablja, izogiblje se napakam, zmore pa tudi ločevati pomembne razlike med diverznimi položajnimi tipi.

(2) PPR hitro izloča raziskovalne smeri, ki so dokončno nekoristne. Tako oklesti dovolj zgodaj +slepe drevorede+ in pridobi čas, ki bi bil potreben za odločanje o brezplodnih razredih možnosti iz prejšnjih raziskav.

(3) PPR izloča redundanco z enkratnim izračunom reči in uporablja kasneje rezultate, če je to potrebno.

(4) PPR pospešuje svoje izračunavanje, kar pomeni v primeru sistema znanja, da povečuje kakovost svojega prevajanja in uporablja hitrejše (zmogljivejše) računalnike.

(5) PPR izrablja tista diverzna telesa znanja, ki bi lahko kaj prispevala k rešitvi problema. Tako uporablja neodvisna izvedeniška telesa pri redukciji večumnosti in izloča šumne vire, ali pa uporablja baze znanja komplementarnih področij in uporablja katerekoli tehnike in hevristične obdelave, ki so za rešitev danega problema najustreznejše.

(6) PPR analizira problem na več načinov, od visoke, abstraktne ravnine do nizke, specifične ravnine.

Pri večini zapletenih problemov se mora problemski reševalnik gibati po abstraktnih ravninah; te ravnine lahko omogočajo vpogled na poljubno ravnino z obsežno dodatno analizo. Primeri takega vpogleda vključujejo npr. razpoznanje, da ima trenutni problem enako obliko kot jo je imel preje reševani, zaznavanje, da neka problemska zahteva izloča iz obravnave vse kandidate razen dveh, ali predočbo, da vsebuje slika pravokotne, vodoravne in navpične linijske segmente, ki predlaga, da gre za objekt, ki ga je naredil človek.

Težavnost naloge problemskega reševanja narašča na štiri načine:

(1) Problemski reševalnik ne razpolaga z natančnimi podatkovnimi viri ali znanjem, ki bi lahko delovalo brez napak. Te pomanjkljivosti lahko povzročijo raziskavo velikega števila nepravilnih poti.

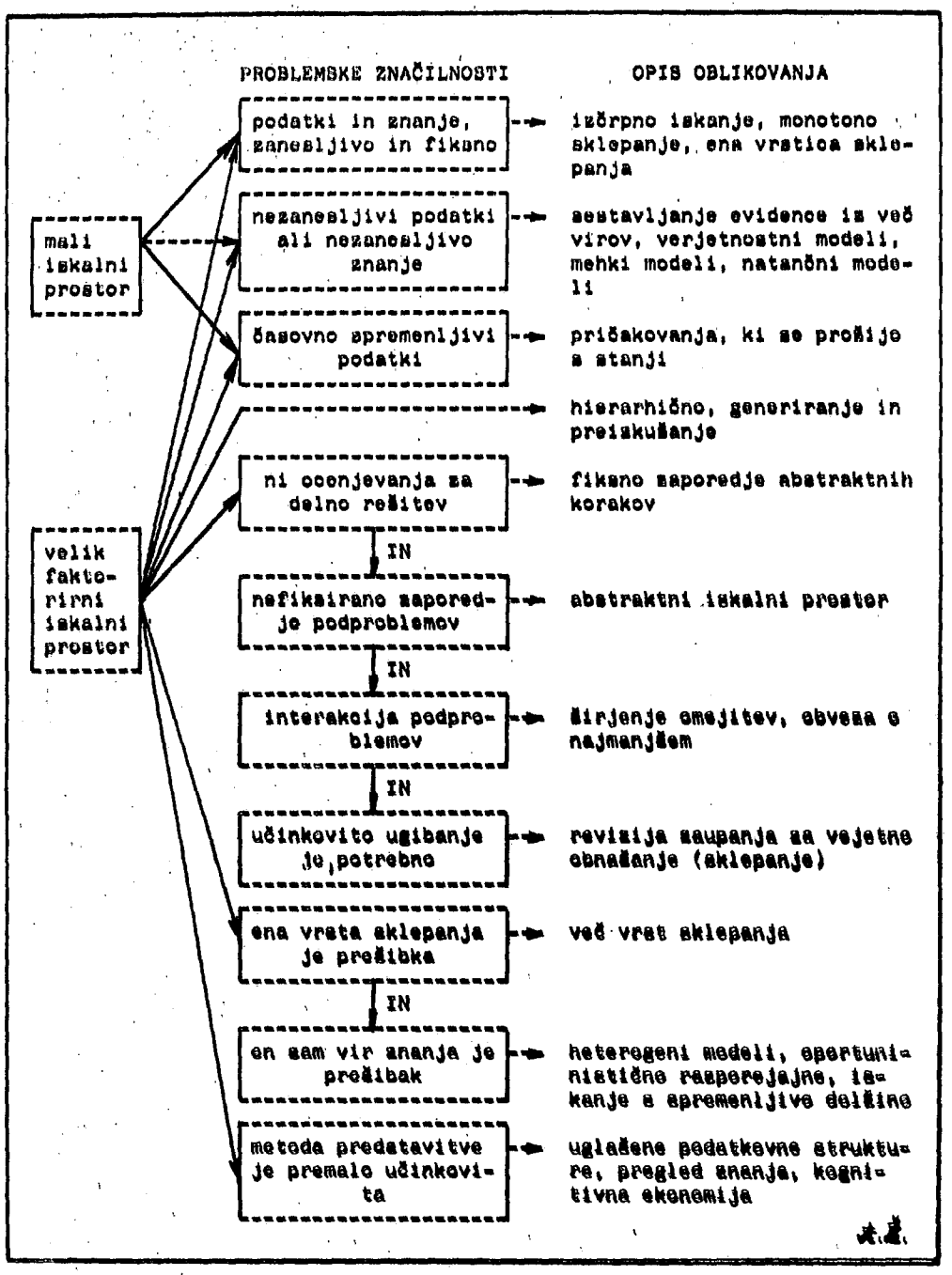
(2) Problemski reševalnik mora pospeševati svoje obnašanje, ko se podatki spreminjajo dinamično, utemeljevati mora svoje odločitve na pričakovanjih prihodnosti in popravljati mora svoje odločitve, ko trenutni podatki kažejo na nepravilnost prejšnjih predpostavk.

(3) Naloga je seveda težja, če je treba upoštevati več možnosti. Vendar je v vrsti uporab težko vnaprej ovrednotiti obseg iskalnega prostora in poiskati alternativne formulacije iskalnega prostora, ki bi kar najbolj poenostavile problem.

(4) Problemski reševalnik, ki mora uporabljati zapletene in časovno zamudne metode pri izločanju alternativ iz obravnave, deluje manj učinkovito od reševalnika, ki ima enako učinkovita toda enostavnejša in cenejša merila (merjenja).

6.2. Organizacija in oblikovanje sistema znanja

Za razliko od navadnih aplikacij obdelave podatkov pa sodobni sistemi znanja ne poznajo takih specifičnih modelov, kot so oblike tipičnih obnavljalnih mojstrskih zbirk ali oblik vhod-obdelava-izhod, ki so pogostne v komercialni obdelavi podatkov. Področje tehnike znanja tudi ne pozna skupnih shem za karakterizacijo svojega oblikovanja in sistemov. Pri oblikovanju sistemov znanja se izkušeni tehniki znanja trdno držijo nekaterih splošnih načel, ki določajo visoke arhitekturne lastnosti sistemov znanja in ki omogočajo, da bo opravljanje njihovih nalog učinkovito. Pri določanju primernega oblikovanja sistemov znanja postavljajo ta načela vprašanja o vrsti zapletenosti problemskega reševanja neke naloge in o vrsti razpoložljivega znanja za hevristično problemsko reševanje. Slika 3 prikazujr vrsto najbolje obvladanih načel oblikovanja.



Slika 3. Arhitekturni predpisi za gradnje sistemov znanja.

Slika 3 razdeli aplikativne probleme sistemov znanja, ki so karakterizirani s malimi in velikimi iskalnimi prostori, na dva dela. Potem prikaže vsako od teh dveh osnovnih kategorij s naštevanjem dodatnih pridevkov, ki tudi lahko karakterizirajo problem. Npr. pri problemih malih prostorov razlikuje tri prekrivajoče podkategorije, ki temeljijo na vrsti podatkov, ki jih sistem znanja mora obdelati. Če izgleda, da so ti podatki zanesljivi in stalni in da deluje sistemsko znanje zanesljivo, predpisuje slika 3 najbolj tipično arhitekturo sistema znanja: tako, ki opravlja izčrpno iskanje in zasleduje eno vratico sklepanja hkrati, kot je npr. globina-prvo nazajšanje verižnje. Predpisani sistem se lahko obnaša monotono: ne potrebuje na začetku formuliranih ugibanj, ki bi jih bilo potrebno kasneje umakniti. V drugi skrajnosti pa prikazuje slika 3 komplekane probleme s velikimi faktorirnimi (raslemljivimi) iskalnimi prostori, v katerih ena vratica sklepanja ne deluje konsistentno in tudi eno telo znanja nima moči za reševanje vseh zadevnih problemov sistema znanja in je začetna oblika predstavitve znanja premalo učinkovita, da bi bile lahko dosežena potrebna zmogljivostna ravnina.

Oblikovalni principi predpisujejo več sdravil. Sistem znanja mora tako raziskovati in razvijati več obetavnih smeri sklepanja, dokler ne pridobi več gotovosti o dejanski rešitvi. Sistem znanja naj vsebuje več neodvisnih podsistemov, od katerih vsak pripomeva k odločanju na opertunistični osnovi. Najvišji sistem znanja naj vodi pri tem pregled nereženih (visečih) podsistemskih akcij, ki obetajo največji

doprinos k razvijajoči se rešitvi. Sistem znanja bo sledil spremenljivemu številu hkratnih, medseboj tekmujočih alternativnih rešitvenih poti, kjer dejansko število v vsaki točki odraža trenutno negotovost o »najboljši« poti. Sistemi znanja naj bi uporabljali različne napredne tehnike za izboljšanje svoje učinkovitosti. Splošno potrebujejo te tehnike neko vrsto transformacije k začetni predstavitvi znanja in k stroju sklepanja. Ta transformacija lahko vključuje prilagoditev podatkovnih struktur tipom sklepanja, prevajanju znanja v novo strukturo (tako, kot je mreža ali drevo), ki omogoča hitro iskanje ali uporabo dinamičnih metod pri lovljenju vmesnih rezultatov.

Današnji principi oblikovanja sistemov znanja razpolagajo z dovolj visokimi napotki za oblikovalca. Podobni so arhitekturnim principom za gradnjo stavb in komercialno konstrukcijo. Ti principi predlagajo široka izhodišča pri konstrukcijskih nalogah brez specifikacije podrobnosti. Sistemi znanja, ki se gradijo skladno s principi slike 3, bodo dobro prilagojeni svojim okoljem, toda se bodo lahko bistveno razlikovali v svoji nadrobni zgradbi.

6.3. P r o j e k t i r a n o z n a n j e

Vidik projektiranja (načrtovanja, izdelave, razvoja) znanja je lahko hkrati jasen (razumljiv, očiten) in težaven (subtilen, domiseln, prenikav, premeten). To, kar se dozdeva jasno, je v tem, da tehnik znanja prevzema znanje od izvedencev in ga integrira v arhitekturo sistema znanja. Torej obstajajo tehniki, ki gradijo sisteme na temelju komponent znanja. To kar je pri tem težavno, je način, kako sistem znanja uporablja znanje pri reševanju problemov, ta način pa je neposredno odvisen od tega, kako tehnik znanja izbira, predstavlja in integrira znanje v sistem.

Znanje se seveda ne dobiva na lahek način, v gotovih paketih, že pripravljeno za uporabo. Znanje je le beseda, ki jo uporabljamo za vrsto fragmentov razumevanja, ki omogoča ljudem in strojem, da opravljajo zahtevne naloge smotrno in dobro. Npr. razumevanje tehnološkega prenosa omogoča tehničnemu direktorju, da sklepa na različne načine o različnih namenih. Pri oblikovanju programa za tehnološki prenos mora tak direktor izostriti in uporabiti znanje na način, ki je drugačen od načina, s katerim bi ocenjeval nek drug program, oceniti mora potrebna sredstva, napovedati rezultate in analizirati podobnost s prejšnjimi uspešnimi ali neuspešnimi programi. Izgleda, da ima človek splošno razumevanje delovanja reči in današnji sistemi znanja lahko vključujejo znatne količine človekovega znanja pri elektronskem reševanju problemov, kar bi sicer zahtevalo človekovo pamet. Sistemi znanja uporabljajo splošno organizacijo, ki je skladna z visokimi oblikovalnimi predpisi in potem uporabljajo v tem okviru znanje reševanja problemov.

7. Gradnja sistemov znanja

Pri gradnji sistemov znanja opravlja tehnik znanja štiri funkcije, ki jih prikazuje slika 4; te funkcije so izkopavanje (iskanje), ulivanje (modeliranje, oblikovanje, upodabljanje), zbiranje (sestavljanje) in čiščenje (rafiniranje). Ti izrazi so izbrani iz slovarja obdelave redkih kovin in ponazarjajo postopke izbire znanja in njegove proizvodnje. Slika 4 prikazuje tudi tehnične termine za štiri primarne graditeljske aktivnosti in označuje ključne produkte posameznih faz.

Naloge obdelave znanja	Tehnične aktivnosti	Tehnični produkti
Izkopavanje	Pridobivanje znanja	Koncepti in pravila
Ulivanje	Oblikovanje sistema znanja	Predstavitev znanja in okvir
Sestavljanje	Programiranje znanja	Baza znanja in stroj za sklepanje
Čiščenje	Čiščenje znanja	Popravljeni koncepti in pravila

Slika 4. Ključne naloge pri razvoju sistemov znanja

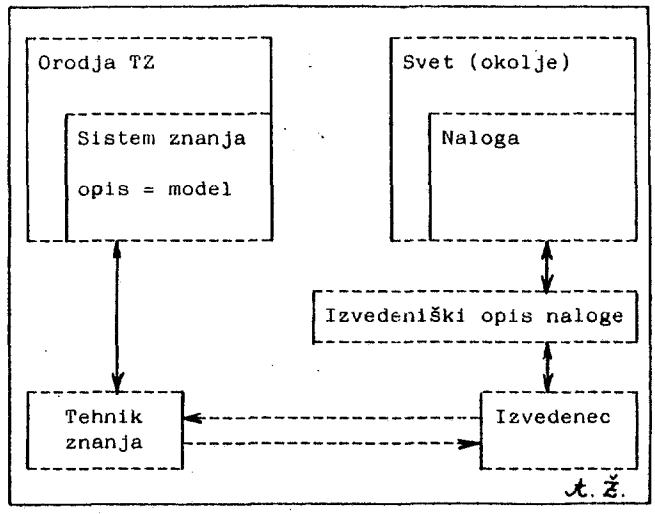
Znanje je podobno kot redka kovina skrito (speče, mirujoče) in nečisto pod površino zavesti. Ko je enkrat izvlečeno, morajo biti njegovi deli tako transformirani, da dobijo komercialno vrednost. Izvlačenje obsega izvabljanje osnovnih konceptov iz problemske domene pri izvedencih ali iz knjig, ko se ugotavljajo člani za opis problemskih stanj in za hevristiko reševanja problemov. Iz te začetne točke se izvlačevanje znanja ali zbiralni postopek nadaljuje, dokler ni izvabljenega toliko znanja za reševanje problemov, da je dosežena t.i. izvedeniška zmogljivost. Hevristična pravila sestavljajo ključni proizvod te dejavnosti.

Oblikovanje sistema znanja producira okvir ali arhitekturo sistema znanja, kot je bilo nakazano. Oblikovalec sistema znanja izbere primerno shemo za predstavitev problemskoreševalnega znanja. Predstavitvene možnosti vključujejo formalno logiko, semantične mreže, hierarhične okvire, aktivne objekte, pravila in procedure, od katerih je vsaka podpirala vsaj enega od prejšnjih naporov razvoja sistema znanja. Ko so tehniki znanja izbrali okvir in predstavitev znanja, se lahko začne programiranje. Oni potem transformirajo človekovo izkustvo v bazo znanja, ki bo napajala stroj za sklepanje.

S človekom razvijajoči današnji sistemi znanja privzemajo obstoječa orodja tehnike znanja, ki vključujejo vnaprej definirani stroj sklepanja, tako da mora programiranje znanja producirati

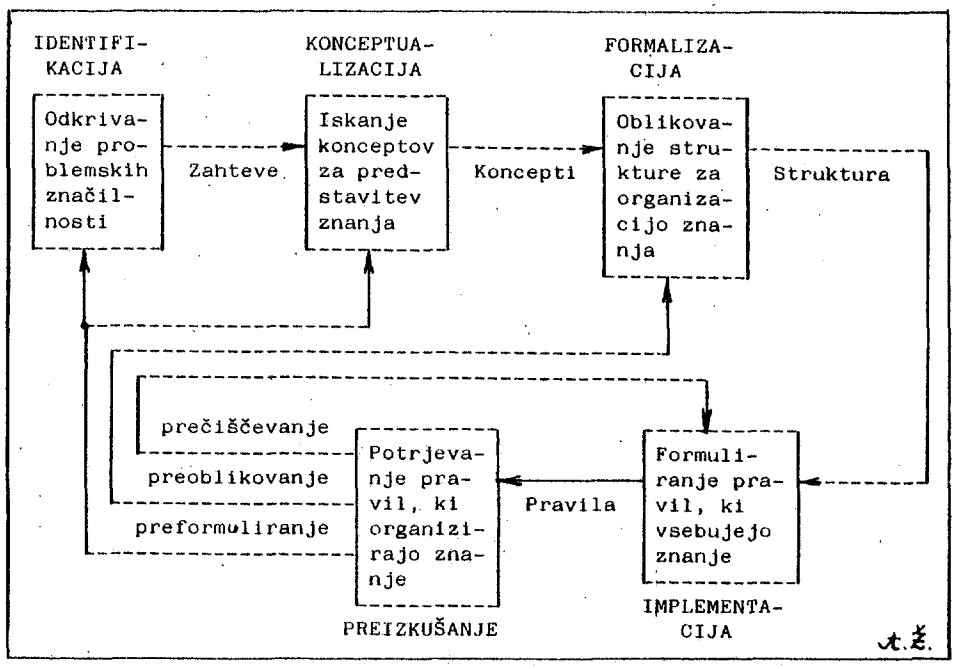
le še bazo znanja. Postopek čiščenja znanja v bazi se nadaljuje dotlej, ko je sistemska zmogljivost dosegla primerno stopnjo. Pri transformaciji nenatančnega razumevanja izvedeniškega obnašanja v hevristična pravila se motita tako izvedenec kot tehnik znanja. Pri tem napačno pojmujeta abstraktne koncepte, nepravilno izražata palčna (prstna) pravila in zanemarjata vrsto podrobnosti, ki pogojujejo veljavnost pravil baze znanja. Splošno velja, da deluje sistem znanja nezadovoljivo na svojem začetku. Ta zmogljivost ne pomeni slabega dela tehnikov; izvedenci opravljajo svoje naloge dobro, ker uporabljajo primerno količino znanja in ne zato, ker morajo razmišljati, kako bi to znanje verbalizirali. S pomočjo tehnike znanja se lahko izvajajo z znanjem intenzivne dejavnosti, ki omogočajo kodificiranje in potrjevanje znanja. Pred pojavom TZ izvedencem ni bilo mogoče izraziti učinkovito njihovega izkustva (znanja) in tako niso mogli določati (ceniti, ocenjevati) znanja empirično. Sistemi znanja pa so omogočili neposredno preizkušanje delovanja znanja in osvetlitev njegovih slabosti in pomnajklivosti. Z odpravo teh pomanjkljivosti lahko izvedenec hitro izboljša bazo znanja. Ta evolucionaren razvoj se najprej približa ravninam človekovih zmogljivosti in jih potem nasplošno preseže.

Slika 5 prikazuje prenos izvedeniškega razumevanja v sistem znanja tehnika znanja, kar predstavlja ključni vidik zbiranja znanja. Ta prenos potrebuje dvosmerno komunikacijo. Najprej se tehnik znanja posvetuje z izvedencem o načinu, kako izvedenec rešuje posamezne probleme in razmišlja o zadevnih predmetih in relacijah (slika označuje te komponente razumevanja kot znanje sveta in znanje naloge). Izvedenec odkrije (odgrne, razodene) nekaj tega znanja pri opisovanju problemskoreševalne naloge, in ga predaja tehniku znanja.



Slika 5. Zbiranje znanja: ob svetovanju izvedenca razvija tehnik znanja sistem znanja za reševanje nalogovnospecifičnih problemov.

Tehnik znanja posluša opis izvedenca, da bi slišal o elementih reševanja problema. Tako kot sistemski analitik oblikuje algoritem za rešitev uporabniškega problema, poskuša tehnik znanja dojeti metodo za reševanje obstoječega problema. Zato izbere orodje TZ in poskuša prilagajati fragmente ekspertize strukturi orodja. Tako mora najprej oblikovati opis načina razmišljanja izvedenca o reševanju problema v dani domeni. Ta opis ponazarja (posnema) ekspertizo izvedenca. Ko je to končno vstavljeno v sistem znanja, začne ta model generirati problemskoreševalno obnašanje, ki ga izvedenec lahko kritizira in izboljšuje. Ta postopek često izboljšuje razumevanje izvedenca o njegovih lastnih izvedeniških zmogljivostih.



Slika 6. Stanja razvoja sistema znanja.

Slika 6 upodablja ponavljajoči, evolucionarni postopek razvoja sistema znanja z osvetljevanjem načinov preizkušanja povratnih povezav sistema znanja k prejšnjim stanjem njegove konstrukcije. Kot kaže slika, lahko preizkušanje izkazuje pomanjkljivosti prejšnjih stanj. In ko napreduje razvoj, je moč opaziti spremembe zahtev, konceptov, organizacijskih struktur in pravil.

(Se nadaljuje v naslednji številki časopisa Informatica)

Slovstvo

(0) F. Hayes-Roth: The Knowledge-Based Expert System: A Tutorial. Computer 17 (1984), No. 9, pp. 11 - 28.

(1) F. Hayes-Roth, D.A.Waterman, D.B.Lenat: Building Expert Systems. Addison-Wesley, 1983

(2) A.Barr, E.A.Feigenbaum: The Handbook of Artificial Intelligence. William Kaufman, Menlo Park, Ca, Vol. 1 (1981), Vol. 2 (1982).

(3) R.K. Lindsay et al.: Applications of Artificial Intelligence for Organic Chemistry: The Dendral Project. McGraw-Hill, New York, 1980.

(4) W.A. Martin, R.J. Fateman: The Macsyma System. Proc. Second Symp. Symbolic and Algebraic Manipulation, 1971, pp. 59 - 75.

(5) E.H Shortliffe: Computer-Based Medical Consultation: Mycin. American Elsvier, New York, 1976.

(6) L.D. Erman et al.: Hearsay II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. Computing Surveys, Vol. 12 (1980), No. 2, pp. 231- 253.

(7) J. McDermott: R1: An Expert in the Computer Systems Domain. Proc. First Annual Nat. Conf. Artificial Intelligence, 1981, pp. 269 - 271.

(8) H.E. Pople, J.D. Myers, R.A. Miller: Dialog Internist: A Model of Diagnostic Logic for Internal Medicine. Proc. IJCAI 4 (1975), pp. 849 - 855.

(9) R. Greiner, D. Lenat: A Representation Language Language. Proc. First Annual Nat. Conf. Artificial Intelligence, Vol. 1, 1980, pp. 165 - 169.

(10) H.P. Nii et al.: Signal-to-Symbol Transformation: HASP_SIAP Case Study. AI Magazine, Vol. 3 (1982), No. 2.

(11) R. Brooks, R. Greiner, T. Binford: The Acronym Model-Based Vision System. Proc. IJCAI 79 (1979), pp. 105 - 113.

(12) L.D. Erman, P.E. London, S.F. Fickas: The Design and an Example Use of Hearsay III. Proc. IJCAI 7 (1981), pp. 409 - 415.

(13) H.P. Nii, N. Aiello: AGE: A Knowledge-Base Program for Building Knowledge-Based Programs. Proc. IJCAI 6 (1979), pp. 645 - 655

A. P. Železnikar

Bibliografija s področja novih računalniških generacij I

Pod tem naslovom bomo objavljali bibliografske vire, ki obravnavajo problematiko, povezano tako ali drugače z novimi računalniškimi generacijami. Pri tem bomo vpeljali osnovne klasifikatorje za posamezna področja, in sicer:

(Ann) Vodilni dokumenti. Navajali bomo dokumente vlad, združenj, podjetij, razne usmeritvene in splošne prispevke.

(Bnn) Pregledni in ozadnji dokumenti (družbeno oklje, jutrišnje perspektive).

(Cnn) Povezava človek_računalnik.

(Dnn) Paralelno procesiranje, nove arhitekture in VLSI.

(Enn) Logično in funkcionalno programiranje.

(Fnn) Izvedeniški sistemi in umetna pamet.

(Gnn) Mreže.

(Hnn) Referenčni (bibliografski) viri.

Pri tem bo nn vsakokratna, zaporedna številka bibliografske navedbe. Navedbi bomo dodali kratek povzetek, na koncu pa bomo napisali še ključne besede, ki jih je moč uporabiti pri iskanju dokumentov v različnih podatkovnih bazah po svetu. Te ključne besede bodo zapisane v angleškem jeziku. Pa začnimo!

A. Vodilni dokumenti

(A1) Jones K. Sparck: Intelligent Knowledge Based Systems: Papers for the Alvey Committee. University of Cambridge Computer Laboratory (June 1982).

Ti članki so bili pripravljeni kot podlaga za Alveyev odbor na zahtevo enega od članov odbora. Ti članki obravnavajo pametne sisteme znanja v petih poglavjih: potreba, definicija, stanje, program in rezultat.

Predlaga se desetletno raziskovanje in razvoj pametnih sistemov znanja, s skupnimi izdatki 67 milijonov funtov, kot sestavni del t.i. Alveyevega programa. Predlog vsebuje dve fazi: uvodno, ki naj pripravi osnovno infrastrukturo za raziskave in razvoj in glavno fazo, ki naj srednjeročno in dolgoročno oblikuje in oceni akademske programe in primerne demonstacijske projekte.

Ključne besede: Alvey programme, IKBS, United Kingdom.

(A2) Research Reports in Japan: a Collection of Recent Research Reports Related to the R and D of the Fifth Generation Computer Systems. Japan Information Processing Development Center (Fall 1981).

To je zbirka 38 poročil z naslednjo problematiko: izpelava formalne specifikacije problema iz njegovega opisa v naravnem jeziku; izpeljava programa iz njegove formalne specifikacije; vidiki logičnega programiranja (12 člankov); strojna arhitektura (9 člankov); razpoznavanje govora (2 članka); jeziki za predstavitev znanja; strojno prevajanje; mrežno usmerjeni operacijski sistemi.

Ključne besede: Collection of Papers, FCGS Project, Japan.

(A3) T. Moto-Oka (Editor): Fifth Generation Computer Systems: Prceedings of the International Conference on Fifth Generation Computer Systems. Tokyo, Japan, October 19-22, 1981. North Holland Publishing Co. (1982).

To je zbornik mednarodne konference, ki je bila v Tokiju od 19. do 22. oktobra 1981, organiziral pa jo je Japan Information Processing Development Center (JIPDEC). Na tej konferenci je bil objavljen program japonske pete računalniške generacije. Zbornik vsebuje 18 člankov, ki so razvrščeni pod naslovi Uvodni govor, Pregledno poročilo, Načrt raziskav informacijske obdelave znanja, Načrt raziskav arhitekture, Povabljeni referati - informacijska obdelava znanja in Povabljeni referati - arhitektura.

Ključne besede: Collection of Papers, FGCS Project, Japan.

(A4) G.G. Scarott (Editor): The Fifth Generation Computer Project: State of the Art Report. Pergamon Infotech Ltd. (1983).

To poročilo je razdeljeno na tri dele: povabljeni referati, analiza in bibliografija. Povabljeni referati preučujejo različne vidike projekta pete generacije računalnikov. Sekcija za analizo obravnava bistvene prednosti projekta pete generacije in podaja uravnoteženo analizo stanja napredovanja pete generacije. Analiza se začenja z upoštevanjem pomena informacije v človekovih dejavnostih in prikazuje stanje informacijske tehnike. Navedena bibliografija je skrbno izbran pregled objavljenih del s področja pete generacije.

Ključne besede: Bibliography, Collection of Papers, FGCS Project, Japan, Social Context

(A5) Intelligent Knowledge Based Systems: A Programme for Action in the UK. Alvey Directorate (August 1983).

Poročilo sestavljajo trije deli in je intenzivna študija posebne komisije Alveyevega programa vlade ZK. Študija je akademska in industrijska ekspertiza in je bila na določen način upoštevana v strategiji Alveyevega programa za področje sistemov, ki temeljijo na znanju.

Prvi del je glavno poročilo in oblikuje strategijo predloženega programa. Drugi del vsebuje vrsto poročil za posamezna podpodročja. Preostali del poročila vsebuje dodatke k prvemu delu poročila.

Ključne besede: Alvey Programme, IKBS, United Kingdom.

(A6) Outline of Research and Development Plans for Fifth Generation Computer Systems. ICOT - Institute for New Generation Computer Technology, Tokyo (May 1982).

To poročilo je povzetek japonskih raziskovalnih in razvojnih načrtov za projekt pete računalniške generacije, ki obravnava socialno in tehnično ozadje projekta, raziskovalna področja in cilje, splošne raziskovalne in razvojne načrte in pripadajočo filozofijo za mednarodno sodelovanje.

Ključne besede: FGCS Project, International Relations, Japan, Social Context.

(A7) Outline of Research and Development Plans for Fifth Generation Computer Systems (Second Edition). ICOT - Institute for New Generation Computer Technology, Tokyo (April 1983, with Supplement dated September 1983).

Ta prispevek je druga izdaja (A6) z bistveno spremenjeno vsebino. Prikazuje se sklepanje o napredovanju projekta pete računalniške generacije, identificirajo se funkcije takega sistema, kot sta npr. reševanje problemov in sklepanje, baza znanja, pametna povezava in pametno programiranje in seveda inovativna materialna (aparaturna) arhitektura in programska oprema za doseganje opisanih funkcij. Opisani so načrti za splošno in začetno fazo, za doseganje projektnih ciljev.

Dodatek nosi naslov Report of RGCS Projects Research Activities, 1982 in je zbirnik projektnih dejavnosti za leto 1982.

Ključne besede: Architecture, FGCS Project, Inference, Intelligent Interface, Japan, Knowledge Base, Problem Solving.

(A8) Proceedings of Research Area Rewiev Meeting in Intelligent Knowledge-Based System. Science and Engineering Research Council (1982).

Zbornik je pregled dvodnevnega londonskega srečanja v septembru 1982, na katerem se je razpravljajlo o predlogu posebnega programa sistemov, ki temeljijo na znanju. Zbornik prinaša poročila razprav o predstavitvi znanja, sklepanju, naravnem jeziku, vidni in objektni manipulaciji, povezavi človek-stroj, izvedeniških sistemih, strojih in programiranju, skupni bazi, raziskovalnih programih in projektih, infrastrukturnih zahtevah, šolanju itd.

Ključne besede: Education and Training Implications, Expert Systems, IKBS, Inference, Knowledge Representation, Man-Machine Interface, Natural Language, UK, Vision.

(Se nadaljuje v naslednji številki časopisa Informatica.)

A. P. Železnikar

Vabilo k sodelovanju

Posvetovanje in seminarji Informatica '85
Nova Gorica, 24.–27. september 1985

Posvetovanje

18. jugoslovansko mednarodno posvetovanje za računalniško tehnologijo in uporabo
Nova Gorica, 24.–27. september 1985

Seminarji

Izbrana poglavja iz računalniške tehnologije in uporabe

Razstava

Razstava računalniške tehnologije, uporabe, literature in drugih računalniških naprav, z mednarodno udeležbo

Roki

1. april 1985 Zadnji rok za sprejem formularja s prijavo in 2 izvodov razširjenega povzetka
15. julij 1985 Zadnji rok za sprejem končnega teksta prispevka

PRIJAVA REFERATA / KRATKEGA REFERATA / STROKOVNEGA POROČILA

Prijavo izpolnite s pisalnim strojem

1. Naslov referata .....
2. Razširjeni povzetek (približno 1000 besed) priložite prijavi.
3. Programsko področje referata (obkrožite ustrezno številko)
   1. Pregled tehnologije in uporabe
   2. Arhitektura in zgradba računalniških sistemov
   3. Upravljanje procesov
   4. Sistemski razvojni pripomočki
   5. Mali poslovni sistemi
   6. Uporaba pri izobraževanju
   7. Osebni računalniki
   8. CAD/CAM mikrosistemi
   9. Umetna inteligenca in roboti
   10. Računalniške mreže
   11. Peta računalniška generacija
4. Razvrstitev referata (obkrožite)
   1. referat (pomembnejše delo)
   2. kratek referat
   3. poročilo

5. Avtorji: .....
   Delovna organizacija: .....
   Ulica: .....
   Poštna številka: ..... Kraj: .....
   Država: .....
   Datum: ..... Podpis: .....

Prijavnica, skupaj z dvema kopijama razširjenega povzetka, mora prispeti najkasneje do 1. aprila 1985 na naslov: Informatica '85, 61116 Ljubljana, p.p. 2

PAPER/SHORT PAPER/ TECHNICAL REPORT REGISTRATION

This application should be typewritten

1. Title of the Paper: .....
2. Extended summary (approximately 1000 words) should be enclosed.
3. Program Category of the Paper (circle appropriate choice)
   1. Survey of Technology and/or Usage
   2. System Architecture
   3. Process Control
   4. Systems Development Tools
   5. Small Business Systems
   6. Usage in Education
   7. Personal Computers
   8. CAD/CAM Systems
   9. Artificial Intelligence and Robots
   10. Computer Networks
   11. Fifth Computer Generation
4. Classification of the Paper (circle appropriate choice)
   1. Paper
   2. Short paper
   3. Technical report

5. Author(s): .....
   Organization: .....
   Street: .....
   Postal Code: ..... City: .....
   Country: .....
   Date: ..... Signature: .....

This application form together with two copies of extended summary must reach the following address before April 1, 1985: Informatica '85, 611.16 Ljubljana, p.p. 2, Yugoslavia

```
================================================
=                                              =
=            Warshallov algoritem              =
=                                              =
================================================

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                              %
%  Informatica UP 20 : Prevajalniki - vaje     %
%  Warshall's Algorithm                        %
%  oktober 1984                                %
%  priredil Anton P. Železnikar                %
%  sistem CP_M, Delta Partner                  %
%  prevajalnik Janus_Ada, verzija 1.5.0        %
%                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

1. Področje uporabe

Matematične relacije lahko predstavimo z Boolovimi matrikami. Elementi teh matrik imajo vrednost 0 ali 1 ali pa tudi false ali true. Vobče bomo imeli kvadratne Boolove matrike razsežnosti n X n.

S e š t e v a n j e dveh Boolovih matrik temelji na operaciji OR (ali) med istoležnimi elementi obeh matrik. Element d(i)(j) vsote matrik d = b + c je določen z adovskim priredilnim stavkom

```
d(i)(j) := b(i)(j) OR c(i)(j);
```

ali pri številski predstavitvi (0, 1) Boolovih matrik z adovskim programskim segmentom

```
IF b(i)(j) = 1 THEN
   d(i)(j) := 1;
ELSE d(i)(j) := c(i)(j); END IF;
```

Člen d(i)(j) p r o d u k t a D = BC je določen z

```
d(i)(j) := b(i)(1) AND c(1)(j) OR
           b(i)(2) AND c(2)(j) OR
           . . . . . OR
           b(i)(n) AND c(n)(j);
```

```
-------------------------------------------------
--      Preizkus Warshallovega algoritma       --
--      Prevajalniki - vaje                    --
-------------------------------------------------

WITH util;

PACKAGE BODY testplus IS

  n: CONSTANT := 50;
  TYPE stolpec IS ARRAY (1 .. n) OF Boolean;
  TYPE Boole_polje IS ARRAY (1 .. n)
                                  OF stolpec;

-- Ta procedura je predmet naše pozornosti:
--*****************************************

PROCEDURE b_plus ( b: IN Boole_polje;
                   a: OUT Boole_polje) IS
  -- Ta procedura predstavlja Warshallov al-
  --    goritem
  i, j, k: integer;
  -- 'n' je globalna spremenljivka

  BEGIN
    FOR i IN 1 .. n LOOP
      FOR j IN 1 .. n LOOP
        a(i)(j) := b(i)(j);
      END LOOP;
    END LOOP;                                  -- (1)
    i := 1;                                    -- (2)
    WHILE i <= n LOOP
      FOR j IN 1 .. n LOOP
        IF a(j)(i) = true THEN
          FOR k IN 1 .. n LOOP
            a(j)(k) := a(j)(k) OR a(i)(k);
          END LOOP;
        END IF;
      END LOOP;                                -- (3)
      i := i + 1;                              -- (4)
    END LOOP;                                  -- (5)
  END b_plus;
--*****************************************

PROCEDURE matr_izp ( m,l: IN integer;
                     e: IN Boole_polje ) IS
  i, j: integer;
  BEGIN
    new_line; put( "Boolova matrika " );
    put(l,2); put( ":" ); new_line;
    FOR i IN 1 .. m LOOP
      FOR j IN 1 .. m LOOP
        IF e(i)(j) = true THEN
          put( " 1" );
        ELSE put( " 0" ); END IF;
      END LOOP;
      new_line;
    END LOOP;
    new_line;
  END matr_izp;
-------------------------------------------------

i, j: integer;
c, d: Boole_polje;

BEGIN
  FOR i IN 1 .. n LOOP
    FOR j IN 1 .. n LOOP
      c(i)(j) := false; d(i)(j) := false;
    END LOOP;
  END LOOP;

  -- Inicializacija vzorčne matrike :
  c(1)(1) := true; c(1)(2) := true;
  c(2)(4) := true; c(3)(5) := true;
  c(4)(2) := true;

  b_plus(c, d);

  matr_izp(5, 1, c);  matr_izp(5, 2, d);
END testplus;
```

Lista 1. Ta lista predstavlja adovski program, ki je določen z Warshallovim algoritmom (koraki 1, 2, 3, 4, 5 v poglavju 2). Tu je b_plus procedura algoritma, matr_izp pa je izpisna procedura za Boolove matrike.

V primeru številske predstavitve (0, 1) Boolovih matrik pa imamo za operacijo AND adovski programski segment

```
IF b(i)(k) = 0 THEN
  delna_vsota(k) := 0;
ELSE delna_vsota(k) := c(k)(j);
END IF;
```

Vsota Boolovih matrik je asociativna in komutativna, produkt je pa asociativen. Obe operaciji zadoščata distributivnostnemu zakonu.

O b r n i t e v relacije R je določena z obrnitvijo Boolove matrike, ki predstavlja relacijo R.

P r o d u k t dveh relacij nad isto abecedo je določen s produktom Boolovih matrik, ki ti relaciji predstavljata.

Produkt R^n je definiran rekurzivno z RR^{n-1} pri $n > 1$. Odtod dobimo z indukcijo po n, da

Boolova matrika B^n predstavlja relacijo R^n. Tako dobimo tale izrek:

Bodi B Boolova matrika razsežnosti n X n, predstavljajoč relacijo R nad dano abecedo. Tedaj Boolova matrika

$$B^+ = B + BB + BBB + \ldots + B^n$$

predstavlja tranzitivno zaprtje R^+ relacije R.

Uporaba relacij in njihovih predstavitev z Boolovimi matrikami nam omogoča programsko določevanje novih relacijskih (parovnih) množic. Z Warshalovim algoritmom pa določamo množice tipa B^+.

Z gramatikami programirnih jezikov je povezanih več relacij, ki se uporabljajo za preizkušanje določenih gramatičnih lastnosti in zlasti pri navzgornji sintaksni analizi. Takšne relacije so npr.

PRVI, PRVI$^+$, ZADNJI, ZADNJI$^+$, ZNOTRAJ,

ZNOTRAJ$^+$, ZNAK, ZNAK$^+$,
prednostne relacije $\doteq$, $\lessdot$ in $\gtrdot$,
PRVITERM, ZADNJITERM,
relacije operatorske prednosti $\doteq$, $\lessdot$ in $\gtrdot$

itn. (glej slovstvo ((1))).

2. Opis programa

Warshall je razvil učinkovit algoritem za računanje tranzitivnega zaprtja B^+ Boolove matrike B. Koraki tega algoritma so tile:

(1) Postavi A := B; (A je Boolova matrika).

(2) Postavi i := 1;.

(3) Za vse j, če je a(j)(i) = true (ali pri številski predstavitvi = 1), potem za k = 1, 2, 3, ... , n postavi

```
a(j)(k) := a(j)(k) OR a(i)(k);
```

(4) Prištej 1 k i.

(5) Če je j manjše ali enako n, nadaljuj s korakom (3), sicer pa se ustavi.

(Glej vajo 3 podpoglavja 3.7.2 na strani 22 slovstva ((1))).

V listi 1 imamo paket testplus za preizkus procedure b_plus, ki predstavlja Warshallov algoritem. S proceduro matr_izp izpisujemo Boolove matrike. S preizkusnim programom najprej inicializiramo matriki c in d, potem pa v matriko c vstavimo za ustrezne njene elemente vrednosti true. Proceduro b_plus uporabimo potem na matrikah c in d, ki ju nato izpišemo.

Lista 2. Ta lista nastane z izvršitvijo programa testplus, opisanega v listi 1. Prva Boolova matrika predstavlja dano relacijo, druga matrika pa njeno tranzitivno zaprtje. Obe matriki se izpišeta šele potem, ko je bila izračunana matrika zaprtja (glej listo 1 na njenem koncu).

```
13A>testplus

Boolova matrika  1:
 1 1 0 0 0
 0 0 0 1 0
 0 0 0 0 1
 0 1 0 0 0
 0 0 0 0 0


Boolova matrika  2:
 1 1 0 1 0
 0 1 0 1 0
 0 0 0 0 1
 0 1 0 1 0
 0 0 0 0 0
```

3. Izvajanje programa

Rezultati izvajanja programa testplus so prikazani v listi 2, kjer imamo obe Boolovi matriki. Prva matrika ponazarja neko relacijo, druga matrika pa tranzitivno zaprtje te relacije.

Slovstvo

((1)) A.P.Železnikar: Prevajalniki. Univerza Edvarda Kardelja v Ljubljani, Fakulteta za elektrotehniko. Ljubljana 1980 (ponatis).

```
================================================
=                                              =
=     Položaj točke glede na mnogokotnik       =
=                                              =
================================================


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                              %
%  Informatica UP 21                           %
%  Position of a Point Concerning to the       %
%    Given Polygon                             %
%  november 1984                               %
%  priredil Anton P. Železnikar                %
%  sistem CP_M, Delta Partner                  %
%  prevajalnik Janus Ada, verzija 1.5.0        %
%                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

1. Področje uporabe

Položaj točke glede na dani mnogokotnik (poligon) je mogoče ugotavljati v povezavi z različnimi geodetskimi, geometričnimi in matematičnimi nalogami. Pri tem gre za odgovor na vprašanje, ali je dana točka v danem poligonu ali ni.

```
------------------------------------------------
--      Položaj točke glede na mnogokotnik     --
------------------------------------------------

WITH floatio, floatops, util;

PACKAGE BODY point IS

  SUBTYPE real IS long_float;
  m: CONSTANT := 20;
  TYPE polje IS ARRAY (1 .. m) of real;

--  Naslednja procedura je predmet naše  pozor-
--    nosti
--**********************************************

  FUNCTION pointpol ( n: IN integer;
                      x, y: IN polje;
                      x0, y0: IN real )
                            RETURN Boolean IS
    i: integer;
    b: Boolean;
    BEGIN
      b := true;
      FOR i IN 1 .. n LOOP
        IF NOT((y0 <= y(i)) XOR (y0 > y(i+1)))
          THEN
          IF x0-x(i) <
             (y0-y(i))*(x(i+1)-x(i))/
                           (y(i+1)-y(i)) THEN
            b := NOT b;
          END IF;
        END IF;
      END LOOP;
      RETURN NOT b;
    END pointpol;
--**********************************************

  PROCEDURE polygon ( n: OUT integer;
                      x, y: OUT polje ) IS
    --  Ta procedura včita 'n' ogljiščnih koor-
    --    dinat poligona in jih shrani v polji
    --    'x' in 'y'

    i: integer;
    BEGIN
      new_line;
    put("Vstavi število poligonskih ogljišč: ");
      get(n); new_line;
      FOR i IN 1 .. n LOOP
        put("Vstavi koordinati ogljišča ");
        put(i,2); put(": x"); put(i);
        put(" = "); floatio.get(x(i));
        put("                              ");
        put("y"); put(i); put(" = ");
        floatio.get(y(i)); new_line;
      END LOOP;
      x(n+1) := x(1); y(n+1) := y(1);
    END polygon;
------------------------------------------------

  PROCEDURE e_point ( k: IN OUT integer;
                      u, v: IN OUT polje ) IS
    --  Ta procedura prebere 'k' koordinatnih
    --    parov točk in jih shrani v polji 'u'
    --    in 'v'

    BEGIN
      put("Vstavi število točk: ");
      get(k); new_line;
      FOR i IN 1 .. k LOOP
        put("Vstavi koordinati "); put(i,2);
        put(": u"); put(i); put(" = ");
        floatio.get(u(i)); put("          ");
        put("              v"); put(i);
        put(" = "); floatio.get(v(i));
        new_line;
      END LOOP;
    END e_point;
------------------------------------------------

  PROCEDURE line (n: IN integer;
                  ch: IN character ) IS
    i: integer;
    BEGIN
      FOR i IN 1 .. n LOOP
        put(ch);
      END LOOP;
      new_line;
    END line;
------------------------------------------------

---------- Glavni preizkusni program ----------

  aa, n, k, i: integer;
  x, y, u, v: polje;

  BEGIN
    n := 3; x(1) := 0.0; y(1) := 0.0;
            x(2) := 1.0; y(2) := 0.0;
            x(3) := 0.0; y(3) := 1.0;
    k := 2; u(1) := 0.5; v(1) := 0.49999999999;
        u(2) := 0.0; v(2) := 1.00000000000001;

<<start>>
    put("Ali želiš preizkušati lego točk ");
    new_line;
    put("glede na mnogokotnik (da=1/ne=0) ? ");
    get(aa); new_line;
    IF aa /= 1 THEN
      GOTO fin; END IF;
    put("Ali želiš vstaviti koordinate ");
    new_line;
    put("ogljišč mnogokotnika (da=1/ne=0) ? ");
    get(aa); new_line;
    IF aa = 1 THEN
        polygon(n,x,y); END IF;
    put("Ali želiš vstaviti točke (da=1");
    put("/ne=0) ? ");
    get(aa); new_line;
```

Lista 1 (se nadaljuje na naslednji strani). Ta lista prikazuje funkcijo pointpol in pomožne procedure polygon, e_point in line za vstavljanje poligonskih ogljišč, opazovanih točk in za vrstični izpis želenega števila znakov.

Lista 1 (nadaljevanje s prejšnje strani). Glavni preizkusni program uporablja deklarirane procedure in je razumljiv sam po sebi.

```
    IF aa = 1 THEN
      e_point(k,u,v); END IF;
    put("Ali želiš preizkus lege (da=1"");
    put("/ne=0) ? ");
    get(aa); new_line;
    IF aa = 1 THEN
      put("         Oglišča mnogokotnika so:");
      new_line; line(43,'=');
      put("         x(i)                    ");
      put("y(i)"); new_line; line(43,'-');
      FOR i IN 1 .. n LOOP
        floatio.put(x(i)); put("    ");
        floatio.put(y(i)); new_line;
      END LOOP;
      line(43,'='); new_line;
      put("          Položaj točk glede na ");
      put("mnogokotnik:");
      new_line; line(53,'=');
      put("         u(i)                    v(i)");
      put("             rezultat");
      new_line; line(53,'-');
      FOR i IN 1 .. k LOOP
        floatio.put(u(i)); put("   ");
        floatio.put(v(i)); put("    ");
        IF pointpol(n,x,y,u(i),v(i)) THEN
          put("znotraj");
        ELSE put(" zunaj"); END IF;
        new_line;
      END LOOP;
      line(53,'='); new_line;
      GOTO start;
    END IF;
<<fin>> null;
  END point;
```

2. Opis programa

Točke s koordinatami x(i) in y(i) (i = 1, ... , n) naj bodo oglišča prostega, zaključenega mnogokotnika, pri čemer so oglišča oštevilčena v krožnem zaporedju. Naj bosta x0, y0 koordinati točke, ki ne leži na kateri od stranic mnogokotnika. V tem primeru opredeljuje funkcija pointpol na listi 1 lego točke (x0, y0) glede na notranjost (oziroma zunanjost) mnogokotnika. Polji x in y funkcije pointpol morata imeti območje 1 .. n 1.

Glavna procedura programskega paketa liste 1 je boolovska funkcija pointpol, katere vrednost je true, če leži točka v notranjosti mnogokotnika. S proceduro polygon se preberejo koordinate poligonskih (mnogokotniških) oglišč, s proceduro e_point pa koordinate vseh tistih točk, katerih lege glede na mnogokotnik želimo ugotavljati. V glavnem preizkusnem programu imamo še segment, ki inicializira enotin trikotnik kot dani mnogokotnik in dve poskusni točki, katerih lego želimo ugotavljati (glej poglavje o izvajanju programa). Temu segmentu sledi še dialogni segment s klici ustreznih procedur.

Lista 2 (se nadaljuje na naslednji strani). Ta lista prikazuje izvajanje programa iz liste 1 za dva primera. V prvem primeru se uporabi v programu definirani trikotnik kot poligon in v programu definirani točki.

```
13A>point
Ali želiš preizkušati lego točk
glede na mnogokotnik (da=1/ne=0) ? 1

Ali želiš vstaviti koordinate
oglišč mnogokotnika (da=1/ne=0) ? 0

Ali želiš vstaviti točke (da=1/ne=0) ? 0

Ali želiš preizkus lege (da=1/ne=0) ? 1

         Oglišča mnogokotnika so:
=================================================
         x(i)                    y(i)
-------------------------------------------------
 0.00000000000000E+0     0.00000000000000E+0
 1.00000000000000E+0     0.00000000000000E+0
 0.00000000000000E+0     1.00000000000000E+0
=================================================

          Položaj točk glede na mnogokotnik:
=================================================================
         u(i)                    v(i)                  rezultat
-----------------------------------------------------------------
 5.00000000000000E-1   4.99999999990000E-1    znotraj
 0.00000000000000E+0   1.00000000000001E+0    zunaj
=================================================================

Ali želiš preizkušati lego točk
glede na mnogokotnik (da=1/ne=0) ? 1

Ali želiš vstaviti koordinate
oglišč mnogokotnika (da=1/ne=0) ? 1


Vstavi število poligonskih oglišč: 4

Vstavi koordinati oglišča  1: x1 = 0
                              y1 = 0

Vstavi koordinati oglišča  2: x2 = 2
                              y2 = 0

Vstavi koordinati oglišča  3: x3 = 2
                              y3 = 1

Vstavi koordinati oglišča  4: x4 = 1
                              y4 = 1

Ali želiš vstaviti točke (da=1/ne=0) ? 1

Vstavi število točk: 6

Vstavi koordinati  1: u1 = 0.3
                      v1 = 0.3

Vstavi koordinati  2: u2 = 0.8
                      v2 = 0.8

Vstavi koordinati  3: u3 = 0.500000000000001
                      v3 = 0.49999999999999

Vstavi koordinati  4: u4 = 1.5
                      v4 = 1

Vstavi koordinati  5: u5 = 1.9
                      v5 = 0.999999999999999

Vstavi koordinati  6: u6 = 2
                      v6 = 1
```

```
Ali želiš preizkus lege (da=1/ne=0)'? 1

        Oglišča mnogokotnika so:
=============================================================
        x(i)                    y(i)
-------------------------------------------------------------
 0.00000000000000E+0     0.00000000000000E+0
 2.00000000000000E+0     0.00000000000000E+0
 2.00000000000000E+0     1.00000000000000E+0
 1.00000000000000E+0     1.00000000000000E+0
=============================================================

        Položaj točk glede na mnogokotnik:
=============================================================
        u(i)                    v(i)                    rezultat
-------------------------------------------------------------
 3.00000000000000E-1     3.00000000000000E-1     znotraj
 8.00000000000000E-1     8.00000000000000E-1     znotraj
 5.00000000000001E-1     4.99999999999990E-1     znotraj
 1.50000000000000E+0     1.00000000000000E+0     znotraj
 1.90000000000000E+0     9.99999999999999E-1     znotraj
 2.00000000000000E+0     1.00000000000000E+0     zunaj
=============================================================

Ali želiš preizkušati lego točk
glede na mnogokotnik (da=1/ne=0) ? 0
```

Lista 2 (nadaljevanje s prejšnje strani). V drugem primeru vstavimo ogljišča četverokotnika in šest poskusnih točk, katerih lego glede na četverokotnik preizkušamo. Od šestih točk jih je pet znotraj poligona, ena je pa zunaj. T.i. mejne točke so lahko znotraj ali zunaj poligona, saj je njihova lega odvisna od natančnosti izračuna v posameznih primerih.

3. Izvajanje programa

Na listi 2 je prikazano izvajanje programskega paketa point, in sicer za dva primera. Prvi primer se nanaša na programsko inicializacijo, ko imamo dan enotin trikotnik in dve točki (ena je znotraj, druga pa zunaj trikotnika). Drugi primer, ki je interaktiven, obravnava štirikotnik in lego šestih točk.

Stopnjevanje študijskih programov računalniških znanosti

Takoimenovano Gourmanovo poročilo (J. Gourman: The Gourman Report. A Rating of Graduate and Professional Programs in American and Internatioanl Universities, National Education Standards, 1982) obravnava stopnjevanje študijskih programov računalniških znanosti 51 univerz, ki so dosegle oceno med 4,0 in 5,0. To poročilo je zanimivo tudi za naša podjetja, ki želijo izpopolnjevati svoje tehnične in tehnološke delavce na vodilnih ameriških univerzah. Razpredelnica 1 prikazuje ocene po določenih parametrih, kot so

študijski program (prog)
fakultetno poučevanje (pouč)
fakultetne raziskave (razi)
knjižnični viri (viri)

ter skupno oceno (ocena) in vrstni red (red).

Obstaja tudi drugačen vrstni red najboljših oddelkov za računalniške znanosti na ameriških univerzah, ki bi lahko bil tale:

MIT (umetna inteligenca)
Stanford (umetna inteligenca)
Berkeley
UCLA
Carnegie-Mellon (formalni jeziki, računalniška arhitektura)
Texas-Austin
Princeton
Harvard
Cornell
Ohio State (grafika, mreže)
Oregon (podatkovne baze)
Rutgers (umetna inteligenca)

Najobsežnejši in najnovejši vir informacij o podiplomskem študiju v ZDA, ki je pri nas na voljo, je priročnik

Petersons Annual Guides- Graduate Study:
Graduate Programs in Engineering and
Applied Sciences 1984

V njem najdemo seznam vseh oddelkov univerz v ZDA in Kanadi, ki so uradno potrjeni za podiplomski študij računalništva. Seznam vsebuje ključne informacije o posameznih oddelkih, kot so:

naslov, statistični podatki, pogoji za vpis,
pogoji za diplome in šolnina.

O nekaterih oddelkih so na posebnih straneh dodane še podrobnejše informacije. Čeprav v tem priročniku ne obstaja uradno stopnjevanje posameznih oddelkov, je mogoče izluščiti ocene za posamezne oddelke z upoštevanjem

uspešnosti raziskovalnega dela,
strokovnosti izšolanih kadrov in
tudi s številom izdanih knjig.

Najbolj zveneča imena ostajajo prejkoslej MIT, Princeton, Carnegie-Mellon, Stanford, Berkeley itd.

Razpredelnica 1

institucija	red	ocena	prog	pouč	razi	viri
MIT	1	4,94	4,93	4,95	4,94	4,94
Ill-Urbana	2	3	2	3	2	3
Cal-Berkeley	3	1	1	2	0	1
Min-Minneapolis	4	0	0	0	88	0
Wis-Madison	5	4,88	4,88	4,89	4,86	4,88
UCLA	6	6	6	7	4	6
Columbia	7	4	4	6	3	4
Harvard	8	3	2	5	1	3
Pennsylvania	9	1	1	3	0	1
Stanford	10	4,79	4,79	2	4,78	4,76
Mich-Ann Arbor	11	7	7	1	4	5
Carnegie-Mellon	12	5	6	4,79	2	3
Purdue-Lafayette	13	3	4	6	0	1
Cal Tech	14	1	2	4	4,69	0
Yale	15	4,69	1	2	6	4,68
N Y U	16	8	4,69	0	5	6
Tex-Austin	17	6	7	4,69	3	4
Cornell	18	4	5	8	1	3
Northwestern	19	3	3	6	0	1
Penn State-U Park	20	0	0	3	4,58	0
Princeton	21	4,58	4,59	1	6	4,57
Rice	22	5	5	4,59	3	4
Wash-Seattle	23	4	4	8	0	2
Rensselaer-NY	24	2	2	6	4,47	1
Cal-San Diego	25	4,48	0	4	3	4,46
Kansas	26	6	4,49	1	0	3
Suny-Buffalo	27	3	6	4,48	4,37	2
NC-Chapel Hill	28	1	4	5	4	4,39
Iowa-Iowa City	29	4,37	1	3	0	5
Brown	30	5	4,39	0	4,28	3
Cal-Davis	31	3	6	4,38	5	1
Rochester	32	1	3	7	3	4,29
Johns Hopkins	33	4,28	2	5	0	6
Case West.Reserve	34	7	1	3	4,19	5
Mich State	35	6	0	1	8	3
Polyt.Inst.-NY	36	4	4,28	0	6	1
Ind-Bloomington	37	2	6	4,26	5	0
Suny-Stony Brook	38	0	4	5	3	4,18
Virginia	39	4,19	2	3	2	7
Duke	40	7	0	1	1	6
Cal-Irvine	41	6	4,18	4,19	0	5
Iowa State-Ames	42	4	6	7	4,09	4
Missouri-Rolla	43	3	4	6	8	2
Md-College Park	44	1	2	7	4	1
Ohio State	45	0	1	2	6	0
Geogia Tech	46	4,09	0	0	5	4,09
Colorado-Boulder	47	7	4,08	4,08	4	7
Texas A and M	48	6	7	7	3	5
Rutgers-New Brunsw	49	4	5	6	2	4
Utah	50	3	4	5	1	3
Houston	51	2	2	3	0	2

Pogoj za vpis na posamezne oddelke računalniških znanosti je razen priznane diplome druge stopnje še uspešno opravljen splošen GRE preizkus. Večkrat pa zahtevajo posamezni oddelki še strokovni GRE preizkus. Kandidati, katerih materin jezik ni angleščina, morajo opraviti še standardni preizkus angleškega jezika (TOEFL).

Oglejmo si za začetek bistvene podatke 11 dobrih (najboljših) oddelkov za računalniške znanosti:

MIT

(1) Glavna področja raziskav: umetna inteligenca, inteligentni sistemi, avtomatizacija načrtovanja VLSI vezij, računalniški sistemi, mreže-komunikacije, podatkovne baze, jeziki za distribuirane sisteme.
(2) Šolnina: 9600 dolarjev za šolsko leto.
(3) Stanovanje: 2000 dolarjev letno v študentskem domu ali približno 500 dolarjev mesečno v družinskem stanovanju.

Princeton

(1) Glavna področja raziskav: načrtovanje računalniških in programskih sistemov, operacijski sistemi, prevajalniki, učinkovita uporaba računalnikov, baze podatkov, grafika
(2) Šolnina: 9350 dolarjev za šolsko leto
(3) Stanovanje: 200 dolarjev mesečno v študentskem domu ali približno 500 dolarjev mesečno v družinskem stanovanju.

Carnegie-Mellon

(1) Glavna področja raziskav: sodobni programski sistemi, teorija računanja, distribuirano procesiranje, umetna inteligenca.
(2) Šolnina: 8250 dolarjev za šolsko leto.
(3) Stanovanje: ni podatkov.

Harvard

(1) Glavna področja raziskav: operacijski sistemi, sistemsko programiranje, teorija računanja, baze podatkov, programirni jeziki.
(2) Šolnina: 9350 dolarjev za šolsko leto.
(3) Stanovanje: 2000 dolarjev letno v študentskem domu ali približno 700 dolarjev mesečno v družinskem stanovanju.

Cornell

(1) Glavna področja raziskav: programirni jeziki in sistemi, teorija računanja, baze podatkov, numerična analiza.
(2) Šolnina: 8900 dolarjev za šolsko leto.
(3) Stanovanje: 1600 dolarjev letno v študentskem domu ali približno 250 dolarjev mesečno v družinskem stanovanju.

Columbia

(1) Glavna področja raziskav: analiza algoritmov, umetna inteligenca, računalniška arhitektura, kompleksnost, baze podatkov, distribuirano računanje, operacijski sistemi, programski sistemi, načrtovanje VLSI vezij.
(2) Šolnina: 9550 dolarjev za šolsko leto.
(3) Stanovanje: do 3000 dolarjev letno v študentskem domu ali približno 300 dolarjev mesečno v družinskem stanovanju.

Stanford

(1) Glavna področja raziskav: umetna inteligenca, analiza algoritmov, teorija računanja, numerična analiza, sistemi.
(2) Šolnina: 9027 dolarjev za šolsko leto.
(3) Stanovanje: 1500 dolarjev letno v študentskem domu ali približno 450 dolarjev mesečno v družinskem stanovanju.

Berkeley

(1) Glavna področja raziskav: računalniška arhitektura, operacijski sistemi, programirni jeziki, baze podatkov, grafika, CAD-CAM, umetna inteligenca.
(2) Šolnina: 4800 dolarjev za šolsko leto.
(3) Stanovanje: 4000 dolarjev letno v študentskem domu ali nad 400 dolarjev mesečno v družinskem stanovanju.

UCLA

(1) Glavna področja raziskav: operacijski sistemi, računalniške mreže in komunikacije, baze podatkov, grafika, CAD-CAM, programirni jeziki, načrtovanje sistemov, umetna inteligenca.
(2) Šolnina: 3360 dolarjev za šolsko leto.
(3) Stanovanje in hrana: 6500 dolarjev letno v študentskem domu.

Minnesota

(1) Glavna področja raziskav: simulacijske metode za načrtovanje operacijskih sistemov, sortiranje na paralelnih računalnikih, podatkovne strukture za računalniške mreže, računska kompleksnost odločitvenih problemov, metode vizualnega procesiranja.
(2) Šolnina: 7000 dolarjev za šolsko leto.
(3) Stanovanje: 2650 dolarjev letno v študentskem domu ali 240 dolarjev mesečno v družinskem stanovanju.

Texas

(1) Glavna področja raziskav: računalniška arhitektura, sistemi za upravljanje baz podatkov, računalniški in naravni jeziki, operacijski sistemi in sistemsko programiranje, komunikacije, programirna tehnika, CAD-CAM, simulacija, umetna inteligenca.
(2) Šolnina: 1270 dolarjev za šolsko leto.
(3) Stanovanje: 1040 dolarjev letno v študentskem domu.

Ti podatki nazorno kažejo, kakšni so lahko stroški, ki izvirajo iz šolnine, stanovanja in seveda iz ostalih potreb (hrana, higiena itd.).

B.Blatnik, A.P.Železnikar

Podiplomski študij računalništva na ameriški univerzi

Univerza University of Kansas v Lawrenceu, v nadaljnjem tekstu KU, je ena izmed boljših ameriških univerz, potrjenih za podiplomski študij računalništva. Podiplomski študij je na vseh ameriških univerzah podobno organiziran, zato ga lahko do precejšnje mere spoznamo na primeru KU.

Študijsko leto sestavljajo trije semestri, jesenski, pomladni in poletni. V vsakem semestru je za vpis na voljo približno polovica predmetov s seznama vseh možnih. Predmeti, po katerih je veliko povpraševanje, se predavajo vsak semester.

Ob vpisu si študent izbere nekaj predmetov. Pri tem mora upoštevati določene smernice, tako da je rezultat študija določen profil kadra. Pomembno vlogo igrajo tudi pogoji za vpis vsakega posameznega predmeta, to je potrebno predznanje. Pri izbiri predmetov in s tem profila lahko študentu pomaga profesor svetovalec. Isti profesor ni nujno kasneje tudi strokovni svetovalec (mentor).

Vsak predmet velja določeno število ur (credit hours, credits). To pomeni, da si študent pridobi to število ur, ko predmet uspešno konča. Določeno število akumuliranih ur je eden izmed pogojev tako za magisterij kot za doktorat. Število ur nekega predmeta praktično pomeni tedensko število ur predavanj za ta predmet. Približno štirikrat toliko časa pa študent dodatno porabi za utrjevanje in poglablanje materije, to je za izdelavo domačih nalog in delo na projektih. Projekti so lahko tudi skupinski. Tako domače naloge kot projekte je potrebno končati in oddati v roku, sicer so razveljavljeni.

Predmeti so označeni po težavnostnih stopnjah. Samo za podiplomce so namenjeni tisti z oznako nad 700, predmete z oznako med 600 in 699 pa lahko vpišejo tudi študenti, ki so pred koncem visokošolskega študija.

Izpitni roki, tako za teste kot končni izpit, so znani že pred začetkom semestra. Končni izpit je nekaj dni po zaključku predavanj. Alternativ ni, razen v izjemnih primerih. Uspešnost se oceni z ocenami A, B, C, D in F. A je odlično, F pa nezadostno. Ocena ni samo rezultat testov in izpita, veliko vlogo igrajo tudi uspešno končane domače naloge in projekti. Tipična razdelitev je naslednja: 15% domače naloge, 35% projekti, 30% dva testa med semestrom, 20% končni izpit.

Vpis

Podiplomski študij računalništva na KU je odprt za vse, katerih reference kažejo sposobnost opravljanja zahtevnejših del v računalništvu. Da je program na široko odprt in hkrati ustreza določenim standardom, morajo imeti kandidati za vpis naslednje predznanje:

(1) Znanje računalništva, primerljivo tistemu, ki se ga dobi na KU pri predmetih: C S 200, C S 210, C S 211, C S 300, C S 400, C S 410, C S 510 in MATH 526. To so temeljni računalniški predmeti.

(2) Znanje diferencialnega in integralnega računa ter najmanj dveh izmed naslednjih tem: linearna algebra, diferencialne enačbe, matematična logika, abstraktna algebra, numerična analiza, verjetnostni račun in statistika.

(3) Poznavanje računalništvu sorodnih znanosti, ki ga kandidat lahko dobi pri štirih zahtevnejših preddiplomskih predmetih na področju elektrotehnike, ekonomije, lingvistike, matematike, fizike in statistike.

Mnogi študentje so si pridobili nekaj tega znanja ob delu ali iz drugih virov. Tudi študentje, ki obvladajo samo del zahtevanega znanja, se lahko vpišejo pod pogojem, da bodo najprej nadoknadili zamujeno in opravili diferencialne izpite. Študentje, ki nimajo dovolj izkušenj v računalništvu, lahko vpišejo poletne tečaje, ki jim pomagajo premostiti razlike v izobrazbi. Vsako poletje (poletni semester) so na voljo vsi osnovni računalniški predmeti. Predno študentje začnejo z rednim podiplomskim delom, dobijo individualno podrobna navodila, kako se naj na to delo pripravijo, to je, na katerih področjih se naj predhodno še sami izpopolnijo.

Prijavi za vpis na podiplomski študij mora biti priložen še dokaz sposobnosti za podiplomsko raziskovalno delo, za kar se šteje dovolj visok rezultat na GRE (Gradual Record Examination) testu. Študentje, katerih materin jezik ni angleščina, morajo dovolj dobro opraviti še test angleškega jezika TOEFL (Test of English as a Foreign Language). Tujci, ki so že diplomirali na kakšni ameriški visoki šoli, so testa TOEFL oproščeni.

Magistrski študij

Za pridobitev naslova magister računalniških znanosti mora kandidat izpolniti naslednje pogoje:

(1) Imeti skupno 30 priznanih ur, od tega najmanj 18 z oznako nad 700. Najmanj 12 od teh 18 ur mora biti iz računalništva.

(2) Imeti široko obzorje v računalništvu, to je obvladati osnove vseh pomembnejših podpodročij računalništva.

(3) Opraviti raziskovalno delo in izdelati tezo; za to je potrebno vpisati najmanj tri ure. Študent, ki ne izdela teze, pripravi eno ali več poročil o raziskovalnem delu, ki se obravnavajo na seminarjih ali v tečajih za posebne probleme.

(4) Uspešno opraviti zaključni pregledni izpit iz glavnih podpodročij računalništva in zagovor magistrske teze ali poročila o raziskovalni nalogi.

Celoten program študija vsakega študenta mora potrditi podiplomska študijska komisija in oddelek. Študent, ki se vključi v program posebno dobro pripravljen in ima vseskozi superiorno povprečje, lahko prosi, da mu zgoraj omenjena komisija dovoli diplomirati z najmanj 24 urami.

Vsak magistrski kandidat mora izbrati predmete tako, da zadovolji takozvane zahteve po področjih. Zahteve po področjih so vpeljali zato, da bi zagotovili tako globino kot širino študija, ne glede na kombinacijo predmetov, ki jo študent izbere. Vsako področje je skupina predmetov, področja pa so urejena takole:

A. Teorija računalništva (dve področji: Avtomati in formalni jeziki, Izračunljivost in kompleksnost)
B. Programirni jeziki in operacijski sistemi (dve področji: Operacijski sistemi in organizacija računalnikov, Osnove in metodologije programiranja)
C. Ostalo (standardno na razpolago osem področij: Računalništvo v humanitarnih vedah, Numerična analiza, Umetna inteligenca, Simulacija in modeliranje, Grafika, Upravljanje podatkovnih baz, Računalniške mreže in komunikacije, Analiza naravnih jezikov)
X. Področje po posebnem dogovoru

Študent mora izbrati dve glavni in dve stranski področji, vključno najmanj eno področje iz A, eno iz B in eno iz C ali X. Izbiro področja X mora potrditi oddlelek.

Na glavnem področju mora študent izbrati dva predmeta, in izdelati oba z oceno A ali enega z A in enega z B. Na stranskem področju izbere študent en predmet, ki ga mora končati z oceno A ali B. Če se hoče študent izogniti ponovnemu vpisu predmeta za boljšo oceno, lahko prosi oddelek za izredni izpit. Fakultetno osebje ustreznega področja bo pripravilo in ocenilo izpit.

Področju je lahko zadoščeno tudi, ne da bi študent vpisal predmet na KU. Na takšne primere naletimo pri študentih, ki so že prej delali na podiplomskem programu kakšne druge univerze ali drugega oddelka na KU (navadno področje X). Seveda morajo izjemo dovoliti in potrditi fakultetni strokovnjaki na ustreznem področju.

Računalniški predmeti

V nadaljevanju je podan seznam vseh možnih podipomskih računalniških predmetov na KU. Nekateri predmeti so podrobneje razčlenjeni, ponekod pa navajam tudi pogoje za vpis - tam, kjer bralec lahko razpozna pomen oznak. C S pomeni Computer Science, cifra v oklepaju pa število ur.

C S 510 Uvod v teorijo računanja II (3)
C S 520 Matematična logika (3)
C S 602 Procesiranje informacij v COBOLu (3)
C S 610 Diskretne strukture (3)
C S 632 Razpoznavanje in generiranje vzorcev (3)
C S 650 Osnove procesiranja simbolov (3)
C S 660 Podatkovne strukture (3)
C S 662 Programirni jeziki (3)
C S 665 Izgradnja prevajalnikov (3)
C S 670 Organizacija računalnikov (3)
C S 675 Mikroračunalniški sistemi in uporaba (3)
C S 680 Numerično računanje (3)
C S 681 Numerična analiza I (3)
C S 682 Numerična analiza II (3)
C S 690 Posebna tema: ... (1-3)
C S 692 Usmerjeno branje (1-3)

C S 710 Uvod v teorijo avtomatov (3)
Struktura, dekompozicije, preslikave in uporaba sekvenčnih in drugih avtomatov.

C S 711 Uporabna Booleova algebra (3)
Booleova algebra z aplikacijami v množicah, logiki, preklopnih vezjih.

C S 716 Teorija formalnih jezikov I (3)
Generiranje formalnih jezikov z gramatikami, razpoznavanje z avtomati in ekvivalentnost teh formulacij; poudarek na kontekstno svobodnih, deterministični kontekstno svobodnih in regularnih jezikih.

C S 717 Teorija formalnih jezikov II (3)
Kontekstno občutjivi in rekurzivno preštevni jeziki; lastnosti zaprtja, odločljivost in dvoumnost nekaterih skupin jezikov. Pogoj: C S 716.

C S 722 Matematična logika (3)

C S 724 Teorija izračunljivosti (3)
Učinkovita izračunljivost funkcij in množic s stališča Turingovih strojev in drugih računalniških modelov; univerzalni stroji; Church-Turingova teza in formalni dokazi ekvivalence Turingovih strojev, sistemi rekurzivnih enačb, Postovi kanonični sistemi; matematične lastnosti razredov rekurzivnih funkcij, rekurzivne in rekurzivno preštevne množice.

C S 726 Meta teorija programirnih jezikov (3)
Algoritmične in formalne lastnosti programov; teoretični študij programov, pravilnost, dokazi lastnosti programov; modeli programov, programske sheme; subrekurzivni programi. Pogoj: C S 716 ali C S 722, zaželeno pa oba.

C S 730 Umetna inteligenca (3)
Uvod v kreativne sisteme za procesiranje informacij, primeri posameznikovega in koordiniranega človeškega obnašanja ter umetno inteligentnih računalniških programov; elementarna analiza sistemov, simulacijskih metod in hevrističnega programiranja, kot orodje za študij razpoznavnih procesov; izgradnja in vrednotenje simulacijskega modela nekega inteligentnega sistema za procesiranje informacij. Pogoja: C S 660 , C S 662.

C S 735 Avtomatsko dokazovanje izrekov (3)
Pregled računalniških metod za dokazovanje izrekov na nekaterih izbranih področjih; podroben študij mehaničnih procedur za dokazovanje izrekov iz predikatnega računa prvega reda; uporaba teh procedur pri dokazovanju pravilnosti programov, sintezi programov, reševanju problemov, deduktivnem odgovarjanju na vprašanja. Pogoja: C S 730 in znanje matematične logike.

C S 742 Zgodovina računalniške tehnologije in informacijske znanosti (3)
Pomembne ideje, izumi in izumitelji od sedemnajstega stoletja do danes; poseben poudarek izvorom inovacij, uporabnosti računalniškega znanja in razvoju računalništva kot samostojne akademske discipline.

C S 744 Socialne teme v računalništvu (3)
Vpliv uporabe računalnikov na družbo; stanje računalništva kot poklicne panoge; položaj računalnikarjev vis-a-vis industriji, šolstvu, vladi, drugim poklicom in ljudem na splošno; primerjalna analiza računalništva v drugih državah. Pogoja: dva izmed C S 660, C S 662, C S 670, C S 760.

C S 745 Človeški faktorji v računalniških sistemih (3)
Človeški faktorji, ki vplivajo na načrtovanje računalniške materialne in programske opreme ter na določitev poslov in nalog; nadzor zmogljivosti pri generiranju in implementaciji programov in sistemov; karakteristike organizacije, ki vplivajo na sprejemljivost in delovanje sistema. Pogoja: C S 662 , C S 670, priporočen tudi C S 660.

C S 748 Kibernetika (3)
Modeliranje in nadzor velikih populacij in sistemov, s posebnim poudarkom na informacijskih, nadzornih in povratnih procesih; poudarek na računalniških postopkih za modeliranje bio-socialnih, načrtovanih sistemov in sistemov človek-stroj.

C S 750 Računalnik kot orodje za razvoj v humanitarnih in družbenih vedah (3)
Poglavja in problemi, izbrani v skladu z interesi slušateljev. Pogoj: C S 650 ali dovoljenje učitelja.

C S 753 Računalniška lingvistika (3)
Pregled računalniških pristopov k študiju fonologije, morfologije in sintakse; računalniška analiza in sinteza govora, razumevanje govora; vsebinska analiza, zbiranje informacij in druga sorodna področja uporabe morfološke analize. Pogoj: C S 660 ali C S 662 ali C S 670.

C S 754 Računalniška semantika (3)

C S 755 Računalniška stilistika (3)

C S 757 Informacijski sistemi (3)
Zajemanje in generiranje informacij iz naravnih jezikov in alfanumeričnih podatkovnih baz. Poudarek na uporabi v izobraževanju, raziskavah in upravljanju, kakor tudi na vrednotenju sistemov z uporabniškega vidika. Pogoja: C S 660 , C S 753 ali C S 754.

C S 760 Operacijski sistemi I (3)
Osnove načrtovanja in implementacija operacijskih sistemov; koncept procesa, asinhroni paralelni procesi, paralelno programiranje; organizacija in upravljanje glavnega in sekundarnih pomnilnikov; organizacija in upravljanje virtualnega pomnilnika; razporejanje procesov, multiprogramiranje; analitično modeliranje in ocena zmogljivosti operacijskih sistemov; mrežni operacijski sistemi; varnost operacijskih sistemov.

C S 761 Operacijski sistemi II (3)
Paralelizem, sinhronizacija procesov, komuniciranje med procesi; življenje sistema, implementacija dinamičnih struktur, načrtovanje modulov sistema in vmesnikov; ažuriranje in dokumentiranje sistema; primerjava nekaterih operacijskih sistemov; izbrani novi pristopi v načrtovanju operacijskih sistemov. Pogoj: C S 760.

C S 762 Programske strukture (3)
Teoretični in praktični aspekti strukturiranega programiranja in strukturiranja podatkov; odnos med rekurzivnimi in iterativnimi strukturami; odnos med statičnimi-sintaksnimi in dinamičnimi-semantičnimi strukturami; načrtovanje, vzdrževanje in preizkušanje programske opreme. Pogoji: C S 660, C S 662 , C S 665 ali C S 760.

C S 764 Analiza algoritmov (3)
Načrtovanje učinkovitih algoritmov; prirojena kompleksnost raznih konkretnih problemov; izbrani problemi iz sortiranja, izbiranja in iskanja podatkov, manipuliranje z množicami, grafi, razni kombinatorični problemi: Pogoj: C S 660.

C S 766 Sistemi za upravljanje podatkovnih baz (3)
Koncepti celovitosti in neodvisnosti podatkov; logične in fizične strukture podatkovnih baz; vmesniki med uporabnikom in podatkovno bazo; jeziki za definicijo in manipuliranje s podatki; paralelen dostop, ažuriranje, varnost in celovitost; vzdrževanje podatkovnih baz. Pogoji: C S 660, C S 662 in C S 670.

C S 768 Simulacija sistemov (3)
Teorija čakalnih vrst, analiza nekaterih modelov; verifikacija simulacijskih modelov, izbira testne statistike; generiranje psevdo naključnih števil; definiranje in uporaba posebnih jezikov za diskretno simulacijo; praktični primeri.

C S 770 Načrtovanje računalniških sistemov (3)
Študij problemov kot so aritmetično in nearitmetično procesiranje, odkrivanje in procesiranje napak, upravljanje in izkoristek pomnilnika, hierarhije pomnilnikov, naslavljanje, nadzor, procesiranje prekinitev, vhod-izhod, vključno z grafiko; primerjava implementiranih in alternativnih rešitev; izbrani novi pristopi v organizaciji računalnikov.

C S 772 Računalniška grafika (3)
Uvod v materialno in programsko opremo za računalniško grafiko, uporaba računalniške grafike. Pogoji: C S 660, C S 662, C S 670.

C S 775 Simulacija in vrednotenje računalniških sistemov (3)
Podrobno proučevanje simulacije računalniških sistemov, meritve in vrednotenje; študij analitičnih modelov, sintetičnih programov, benchmark testov, simulacija in opazovanje zmogljivosti; izbrani novi pristopi za merjenje zmogljivosti programske in materialne opreme, primerjava implementiranih in alternativnih rešitev. Pogoja: C S 760 in C S 768.

C S 778 Računalniške mreže (3)
Proces izgradnje mrežnih sistemov, vključno ARPA prototipna tehnologija in nadaljnji razvoj; standardi in protokoli; distribuirane funkcije, miniračunalniške in mikroračunalniške konfiguracije; virtualne mreže za znanstvene aplikacije; telekomunikacijski predpisi; akt o svobodni izmenjavi informacij, privatnost, enkripcija, zagotovitev celovitosti; razvoj mrež drugod. Pogoj: C S 760.

C S 780 Numerična analiza linearnih sistemov (3)

C S 781 Numerična funkcionalna analiza (3)

C S 784 Računalniška algebra (3)

C S 785 Optimizacijska teorija: računalniški pristop (3)

C S 790 Nadaljevalne teme: ... (1-3)
Urejeno glede na potrebe podiplomskih študentov.

C S 792 Usmerjeno branje (1-3)
Podrobnejši študij kakšnega področja pod nadzorom mentorja; materija, ki se pri rednih predmetih ne obravnava. Oddelek mora dovoliti vpis.

C S 795 Seminar: ... (1-3)
Diskusije in poročila o najnovejših dosežkih v računalništvu - spremljanje literature.

C S 797 Posebni problemi (1-3)
Podiplomsko raziskovalno delo na posebnih področjih pod vodstvom strokovnjaka.

C S 810 Teorija avtomatov (3)
Formalni računalniški modeli in njihova uporaba, poudarek na algebraičnih in logičnih aspektih.

C S 816 Računska kompleksnost avtomatov (3)

C S 823 Teorija rekurzivnih funkcij (3)

C S 824 Abstraktna računska kompleksnost (3)

C S 830 Umetna inteligenca II (3)

C S 881 Numerično reševanje nelinearnih operatorskih enačb (3)

C S 882 Numerično reševanje diferencialnih enačb (3)

C S 883 Numerično reševanje parcialnih diferencialnih enačb (3)

C S 885 Numerična aproksimacija funkcij (3)

C S 887 Računalniška statistika (3)

C S 890 Podiplomska tema:... (1-3)

C S 895 Poročila o raziskavah - seminar:... (0-3)

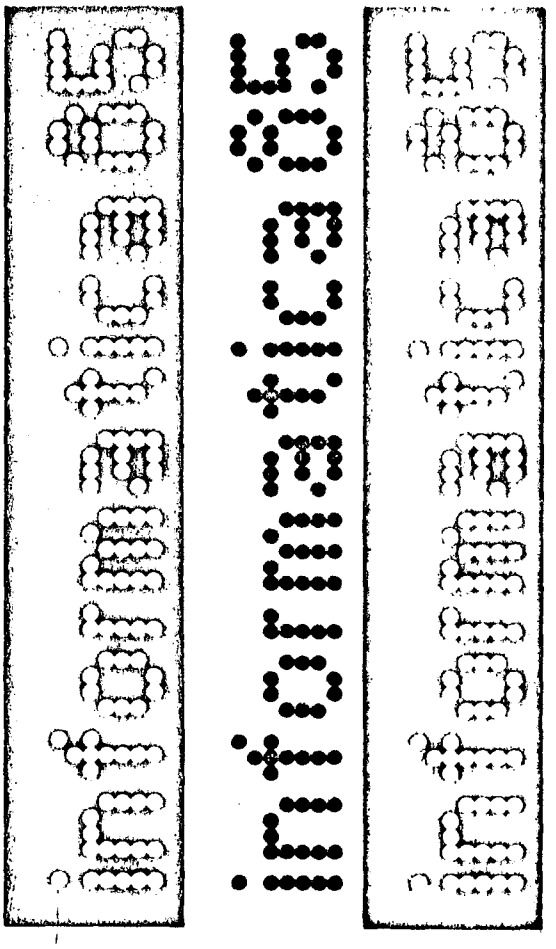
C S 898 Magistrsko poročilo o raziskovalni nalogi (1-6)

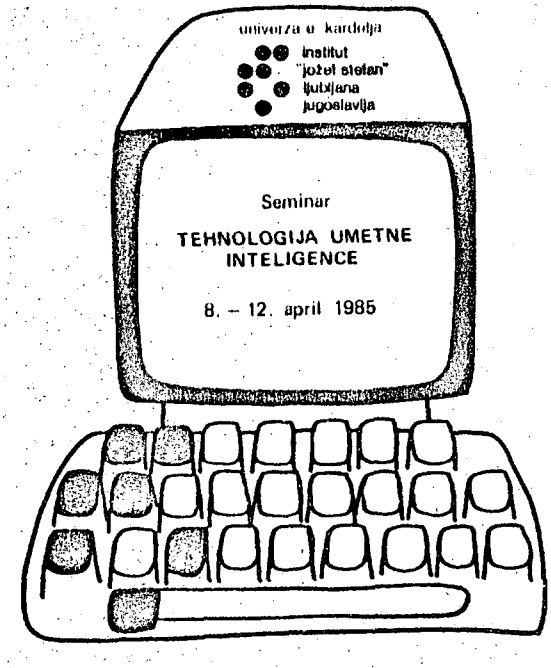
C S 899 Magistrska teza (1-6)

C S 998 Postmagistrska raziskava (1-6)

C S 999 Doktorska disertacija (1-10)

B. Blatnik





Znanstveno – tehnološki seminar

TEHNOLOGIJA UMETNE INTELIGENCE

8. – 12. april 1985

Institut Jožef Stefan, Ljubljana

S tem prispevkom bi radi obvestili bralce revije Informatica o možnosti, da se udeležijo seminarja "Tehnologija umetne inteligence". Seminar organizira Odsek za računalništvo in informatiko Instituta Jožef Stefan v sodelovanju s Slovenskim društvom Informatika. Potekal bo od 8. do 12. aprila 1985 v prostorih Instituta Jožef Stefan. Seminar bo obravnaval sodobne teme umetne inteligence s poudarkom na tekočih projektih umetne inteligence v svetu ter na možnostih uporabe orodij umetne inteligence za izdelavo konkretnih aplikacij v našem razvojnem, proizvodnem, poslovnem in upravnem okolju. Seminar bo podal tudi vpogled v projekt 5. generacije računalnikov in na programski jezik prolog, ki je izbran za jezik te generacije.

Umetna inteligenca in njeni produkti so že dalj časa predmet živega zanimanja raziskovalcev in industrije na Japonskem, v ZDA in Zahodni Evropi, pa tudi pri nas. K razvoju teh znanj v svetu že vrsto let enakopravno prispevata Skupina za umetno inteligenco na Odseku za računalništvo in informatiko Instituta Jožef Stefan in na Fakulteti za elektrotehniko Univerze E. Kardelja v Ljubljani.

Program seminarja

Celotni program seminarja bo razdeljen v osnovni program in dodatni program. Osnovni program bodo sestavljale strokovne predstavitve sodobnih tem umetne inteligence. Dodatni program pa bo voden v obliki šole programiranja v programskem jeziku prolog, ki je izbran kot programski jezik 5. generacije računalnikov. Pri praktičnem delu bo poudarek na aplikacijah v ekspertnih sistemih. Osnovni program bo potekal pretežno dopoldne, dodatni pa popoldne v prostorih Instituta Jožef Stefan.

Osnovni program: sodobne teme umetne inteligence

- Metode in tehnike umetne inteligence
- Ekspertni sistemi
- Tekoči projekti v svetu
- Nekatere aplikacije umetne inteligence
- Projekt 5. generacije računalnikov
- Oprema, potrebna za razvoj ekspertnih sistemov
- Programirni pripomočki za umetno inteligenco (programski jeziki, okolja in orodja)
- Programski jezik prolog
- Vloga umetne inteligence in jezikov 4. generacije v informacijskih sistemih
- Metode umetne inteligence v odločanju
- Predstavitev projektov umetne inteligence pri nas
- Demonstracije delujočih programskih paketov.

PRIJAVNICA ZA SEMINAR "TEHNOLOGIJA UMETNE INTELIGENCE"
8. – 12. april 1985, Institut Jožef Stefan, Jamova 39, Ljubljana

Ime in priimek:

Naslov delovne organizacije:

Telefonska številka:

Prijavljam se za (označite ustrezno okence):

| | celotni program | | osnovni program

Sem sodelavec akademske institucije (označite ustrezno okence):

| | da | | ne

Sem sodelavec sponzorske institucije (označite ustrezno okence):

| | da | | ne

Kotizacijo v znesku (označite ustrezno okence):

| | 35.000 din | | 30.000 din | | 25.000 din | | 20.000 din

bom poravnal na ŽR Instituta Jožef Stefan 50101-603-50272 najkasneje do 1. aprila 1985.

Dodatna informacija:

Udeleženci seminarja bodo predvidoma lahko kosili v menzi Instituta Jožef Stefan (doplačilo ob registraciji). Prosimo vas za informacijo, če ste kandidat za kosila na IJS (označite ustrezno okence):

| | da | | ne

Podpis:

Dodatni program: šola programiranja v prologu

Šola programiranja v prologu bo omogočala udeležencem učenje programiranja v prologu na računalnikih VAX in PDP. Udeleženci bodo razdeljeni v več skupin. Praktično delo bo potekalo pod strokovnim vodstvom delovnih mentorjev.

Okrogla miza: Današnji trenutek umetne inteligence v Jugoslaviji

Udeleženci bodo v organizirani diskusiji razpravljali o stanju umetne inteligence v Jugoslaviji in možnostih uporabe orodij umetne inteligence na konkretnih aplikativnih področjih. K diskusiji bodo zlasti pozvani predstavniki razvojnih, poslovnih in proizvodnih organizacij ter uprave.

Komu je seminar namenjen

Udeležba na sestanku bo zanimiva tako za sodelavce akademskih (raziskovalnih in izobraževalnih) institucij kot za predstavnike iz gospodarstva, družbenih dejavnosti in državne uprave. Ti se bodo seznanili z možnostmi uporabe sodobnih orodij za reševanje nekaterih poslovnih in proizvodnih problemov, ki jih s klasičnimi metodami ni možno kvalitetno rešiti.

Predavatelji

- prof.dr. Ivan Bratko (vodja Skupine za umetno inteligenco na IJS, izredni profesor na FE, predsednik Jugoslovanske sekcije za umetno inteligenco pri združenju ETAN, predstavnik Jugoslavije v ECCAI (European Coordination Committee for Artificial Intelligence) ter eden od direktorjev združenja ISSEK (International School for the Synthesis of Expert Knowledge)
- mag. Matjaž Gams (sodelavec IJS in asistent na FE)
- mag. Marjan Krisper (docent na FE)
- mag. Nada Lavrač (sodelavka IJS)
- mag. Vladislav Rajkovič (predavatelj na VŠOD in sodelavec IJS)
- dr. Marjan Špegel (vodja Odseka za računalništvo in informatiko IJS)
- mag. Peter Tancig (sodelavec IJS in docent na FE).

Delovni mentorji

Sodelavci projektov iz umetne inteligence na IJS in FE.

Kotizacije

- za udeležence celotnega programa: 35.000 din
- za udeležence osnovnega programa (brez šole programiranja v prologu) : 30.000 din.

Popusti

Omogočamo popust 10.000 din naslednjim udeležencem seminarja:
- sodelavcem akademskih institucij
- sodelavcem sponzorskih institucij.

Sponzorstvo

Seminar bo organiziran pod sponzorstvom zainteresiranih jugoslovanskih institucij. Sodelavci sponzorskih institucij bodo deležni ustreznega popusta, imeli bodo prednost pri prijavi na dodatni program ter prednost pri konzultacijah o možnostih uporabe orodij umetne inteligence pri načrtovanju aplikacij za potrebe svoje institucije.

Odobrena podpora

Seminar bodo s podporo omogočile naslednje institucije:
- Institut Jožef Stefan
- Raziskovalna skupnost Slovenije
- Slovensko društvo Informatica
- Organizacije – sponzorji.

Omejitev prijav

Zaradi omejenih računalniških virov bo udeležba v dodatnem programu omejena na 20 udeležencev. Pri prijavah za dodatni program bodo imeli prednost sodelavci sponzorskih organizacij, zatem pa bomo udeležbo v dodatnem programu selekcionirali po kriteriju zaporednega registriranja prijav.

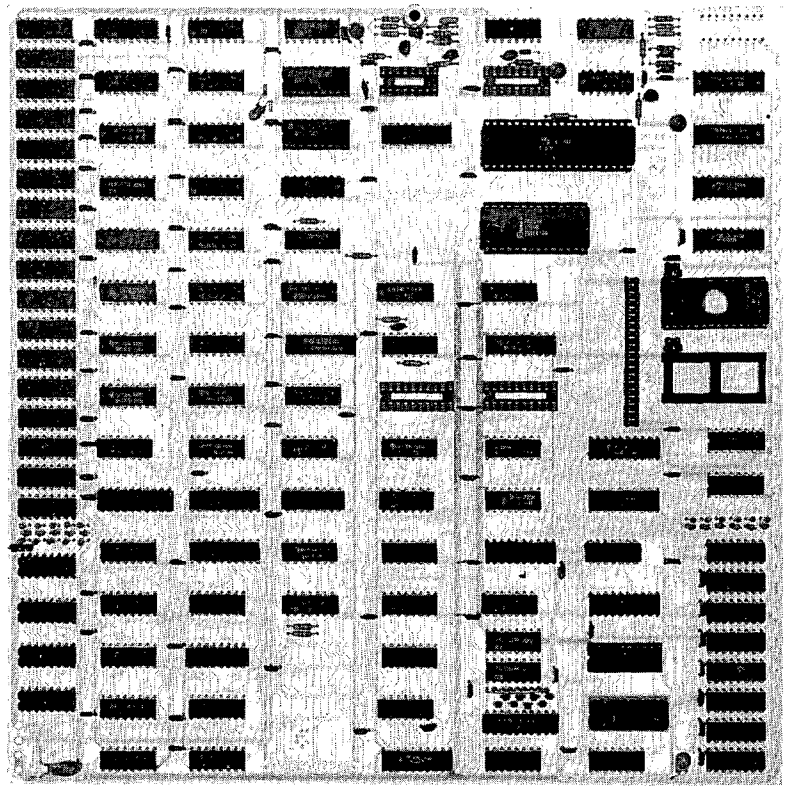
Prijava

Udeleženci se za seminar prijavijo z izpolnitvijo prijavnice, ki mora priti na naslov organizatorja najkasneje do 15. marca 1985. Kotizacije morajo prispeti na žiro račun IJS: 50101-603-50272 najkasneje do 1. aprila 1985. Za nadaljnje informacije lahko kličete (061) 214-399, int. 217 (Nada Lavrač, Matjaž Gams) ali int. 528 (Diana Kobler).

Prijavnico za seminar "TEHNOLOGIJA UMETNE INTELIGENCE" pošljite na naslov:

mag. Nada Lavrač
Odsek za računalništvo in informatiko
Institut Jožef Stefan
Jamova 39
61000 Ljubljana

GRAF–100
GRAFIČNI PROCESOR ZA VIDEOTERMINAL KOPA 1000 (VT100)



LAGRAF–120
GRAFIČNI DODATEK ZA RISANJE NA MATRIČNEM PISALNIKU DEC LA–120