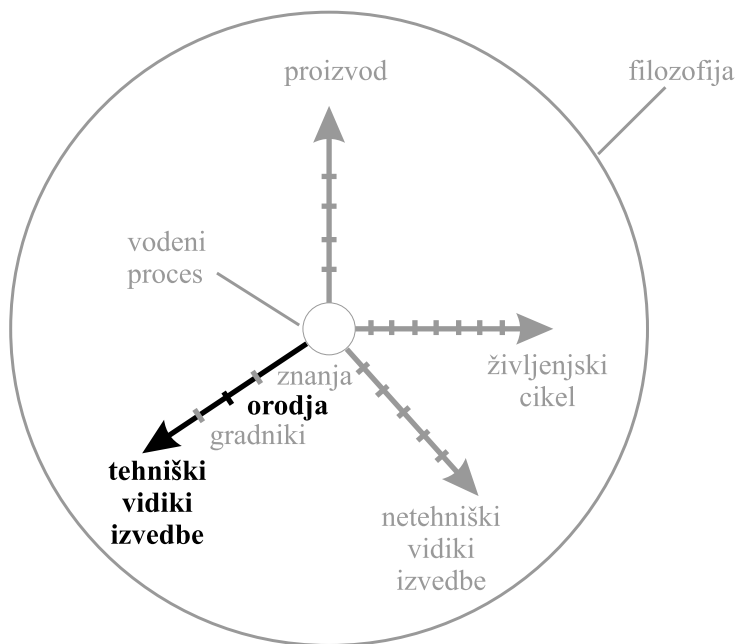


8. Tehniški vidiki izvedbe

Orodja



8.1 Uvod

Poglavje, ki je pred nami je namenjeno opisu nekaterih najbolj pogostih orodij, s katerimi se srečujemo pri načrtovanju, izgradnji in uporabi računalniških sistemov za vodenje.

Prvi del poglavja se nanaša na orodja za načrtovanje funkcij vodenja. To so univerzalna orodja, ki omogočajo načrtovanje na zelo abstraktnem nivoju ter neodvisno od opreme, s katero bomo funkcije implementirali. Orodja so seveda zaradi tega bolj prilagojena metodam oziroma postopkom vodenja, manj pa gradnikom, s katerimi konkretne sisteme realiziramo. To tudi pomeni, da so bolj namenjena načrtovanju in manj izvedbi.

V drugem delu poglavja so predstavljena orodja za sistemsko analizo in razvoj programske opreme. Ta seveda omogočajo učinkovito in kvalitetno realizacijo programske opreme od začetnih do končnih faz njenega življenjskega cikla. V tem okviru opisujemo predvsem univerzalna orodja, ki niso vezana na specifično materialno opremo ali celo specifični gradnik.

Tretji del pa je namenjen predstavitvi orodij za konfiguriranje in implementacijo funkcij vodenja. Tu gre za specifične programske pakete, ki bodisi omogočajo konfiguriranje in parametrisiranje vnaprej pripravljenih funkcij vodenja v okviru določenega programskega paketa, ki teče na določeni računalniški opremi ali pa specifičen razvoj programske opreme za specifičen gradnik (npr. regulator ali krmilnik).

8.2 Orodja za načrtovanje funkcij vodenja

V orodja za načrtovanje funkcij vodenja smo uvrstili simulacijska orodja oziroma simulacijske jezike, ki omogočajo enostavno eksperimentiranje z matematičnimi modeli procesov ter različnimi algoritmi vodenja ter orodja za računalniško podprto načrtovanje vodenja (takoimenovana CACSD orodja), ki omogočajo analizo dinamičnih lastnosti sistemov in sintezo algoritmov za vodenje.

8.2.1 Orodja za simulacijo zveznih dinamičnih sistemov

Simulacijsko orodje je programska oprema ali kombinacija programske in materialne opreme, ki omogoča uporabniku enostaven prenos matematičnega modela v računalniški (simulacijski model) in uporabniško prijazno eksperimentiranje. Pri tem je pomembno, da se lahko uporabnik kot strokovnjak za modeliranje osredotoči na reševanje realnega problema, ne pa na programiranje. Orodje mora biti ustrezno interaktivno, numerično robustno, mora pa omogočati tudi učinkovito dokumentiranje modelov in simulacijskih rezultatov.

Metode, ki smo jih spoznali v podpoglavju 7.2.2 (Metode za simulacijo dinamičnih

sistemov), omogočajo, da s pomočjo simulacijske sheme direktno napišemo program v računalniškem simulacijskem orodju. Če pa želimo sami programirati simulacijo v nekem splošnonamenskem programskem jeziku, (npr. FORTRAN, C, PASCAL, ...) ali če želimo razumeti delovanje simulacijskih orodij, pa je potrebno poznati koncept digitalne simulacije zveznih dinamičnih sistemov, kajti integrator je potrebno realizirati z numeričnim postopkom, paralelni sistem pa računati zaporedno.

8.2.1.1 Koncept digitalne simulacije

Da lahko simuliramo zvezni sistem na digitalnem računalniku, moramo neodvisno spremenljivko simulacije (običajno čas) diskretizirati, tako da diferencialne enačbe postanejo diferenčne enačbe. Ker so vse spremenljivke modela odvisne od neodvisne spremenljivke t , so torej tudi definirane samo v diskretnih vrednostih (trenutkih) neodvisne spremenljivke. Če simulacija poteka s konstantnim prirastkom neodvisne spremenljivke Δt (*računski korak*), lahko trenutno vrednost neodvisne spremenljivke izrazimo kot

$$t_i = t_0 + i\Delta t \quad i = 0, 1, 2, \dots, i_{\max}$$

Torej simulacija prične v trenutku $t = t_0$ (*začetni čas simulacije*) in konča v trenutku $t = t_{\max} = t_0 + i_{\max} \Delta t$ (*končni čas simulacije*). Dogajanju med časoma t_0 in $t_{\max}$ pravimo *simulacijski tek*.

Vsi sistemi (orodja) za digitalno simulacijo so osnovani na zapisu sistema v prostoru stanj, t.j. s sistemom diferencialnih enačb 1. reda (glej podpoglavje Metode za simulacijo dinamičnih sistemov). Uporabniku pa se tega običajno niti ni potrebno zavedati, saj se ustrezni zapis avtomatično vzpostavi. Izhodišče je torej že večkrat predstavljena vektorska diferencialna enačba

$$\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t)) \quad (8.1)$$

Med simulacijo se vsa stanja hranijo v vektorju $\mathbf{x}(t)$, vsi odvodi pa v vektorju $\dot{\mathbf{x}}(t)$.

Pri uporabi indirektnega načina je torej prvi korak že opravljen. Drugi korak pa se realizira z enim vektorskim integratorjem

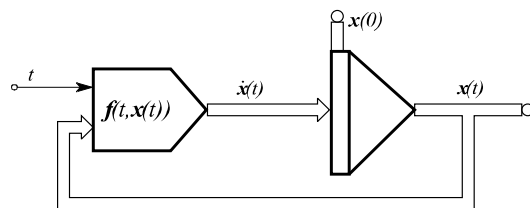
$$\mathbf{x}(t) = \int \dot{\mathbf{x}}(t) dt = \int \mathbf{f}(t, \mathbf{x}(t)) dt \quad (8.2)$$

Z upoštevanjem tretjega koraka indirektnega postopka dobimo simulacijsko shemo, ki jo prikazuje Sl. 8.1.

Da lahko shemo na Sl. 8.1 simuliramo na digitalnem računalniku, mora imeti vsak simulacijski sistem tri pomembne značilnosti:

Zaporedno računanje: Ker splošnonamenski sekvenčni digitalni računalniki ne zmorejo paralelnega računanja kot npr. analogni računalniki, je potrebno vse zanke, ki

jih prikazuje Sl. 8.1, na nek način prekiniti, tako da je možno spremenljivke računati zaporedno. Zanke se prekinijo na izhodih t.i. spominskih blokov (včasih jim pravimo tudi bloki z zakasnitvenim atributom), katerih izhodi so običajno stanja sistema. V splošnem je to lahko vsak blok, ki ima lastnost, da trenutna vrednost njegovega izhoda ni odvisna od vrednosti vhoda v istem trenutku. Najpogosteje so taki bloki integratorji.

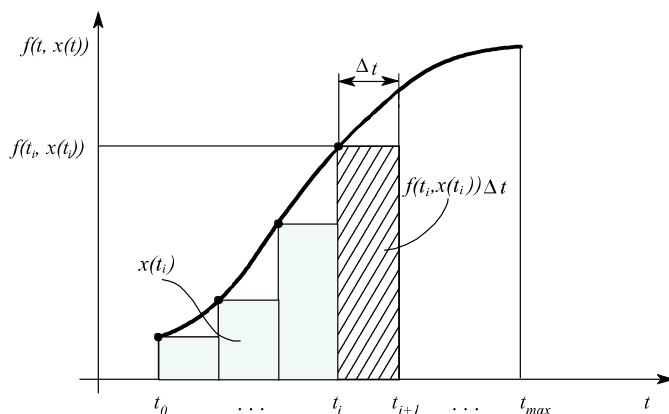


Sl. 8.1. Osnovna simulacijska shema

Vrstni algoritem: Vrstni algoritem mora razvrstiti enačbe (bloke) za izračun odvodov $f(t, \mathbf{x}(t))$, enačba (8.1) tako, da je možno pravilno izračunavanje, t.j. da se vsi vhodi v nek blok izračunajo prej, preden se izračunavajo enačbe (oz. izhodi) tega bloka.

Numerična integracija: Enačbo (8.2) je potrebno rešiti z numeričnim integracijskim postopkom. Integracija je osrednja operacija vsakega simulacijskega orodja. Če ne potrebujemo velike natančnosti, jo je možno realizirati z zelo enostavnim numeričnim postopkom. V modernih numerično izpopolnjenih simulacijskih jezikih pa ti postopki omogočajo veliko natančnost in numerično zanesljivost in so zato zelo kompleksni. Princip numerične integracije najlepše prikazuje najenostavnejši postopek, t.j. Euler-jev algoritem, ki opisuje reševanje enačbe (8.2) v obliki

$$\mathbf{x}(t_{i+1}) = \mathbf{x}(t_i + \Delta t) = \mathbf{x}(t_i) + \mathbf{f}(t_i, \mathbf{x}(t_i)) \Delta t \quad (8.3)$$

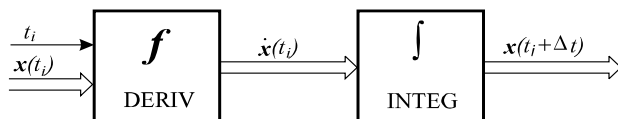


Sl. 8.2. Integracija z Euler-jevo metodo

Metoda predpostavlja stopničasto aproksimacijo funkcije $f(t, \mathbf{x}(t))$, ustrezen integral pa je vsota pravokotnikov pod krivuljo $f(t, \mathbf{x}(t))$. Postopek prikazuje Sl. 8.2.

Integracijska metoda torej uporablja trenutne vrednosti stanj $\mathbf{x}(t_i)$ in odvodov $f(t_i, \mathbf{x}(t_i))$ za ovrednotenje stanj za naslednji računski korak $\mathbf{x}(t_i + \Delta t)$.

Če zanke sistema, ki ga prikazuje Sl. 8.1, prekinemo na izhodu vektorskega integratorja, in če zamenjamo zvezno integracijo z numeričnim postopkom, dobimo shemo, ki jo prikazuje Sl. 8.3. Ta shema jasno prikazuje koncept digitalne simulacije zveznih dinamičnih sistemov.



Sl. 8.3. Osnovni koncept digitalne simulacije

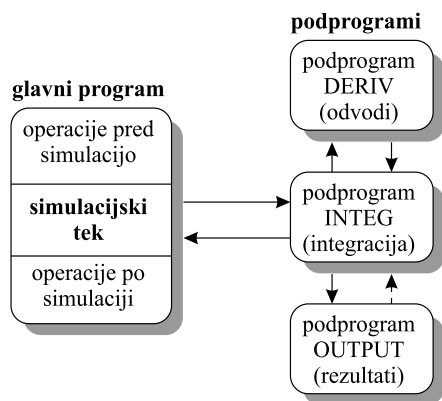
Na začetku simulacijskega teka se vedno izvede inicializacija, ki vključuje tudi ovrednotenje začetnih vrednosti stanj ($\mathbf{x}(t_0)$).

Simulacijski algoritem lahko opišemo z dvokoračno iteracijo v vsakem računskem koraku. V prvem koraku se izračunajo vsi odvodi spremenljivk stanja ($\dot{\mathbf{x}}(t_i)$) iz trenutnih vrednosti spremenljivk stanja ($\mathbf{x}(t_i)$) in iz vrednosti neodvisne spremenljivke simulacije (t_i). Odvodi se izračunajo v podprogramu, ki ga bomo imenovali **DERIV** in vključuje izračun funkcije $f(\mathbf{x}, t)$ oz. ustrezno razvrščene enačbe modela. V drugem koraku se v podprogramu, ki ga bomo imenovali **INTEG**, izvrši integracijski postopek, ki izračuna stanja za naslednji računski korak ($\mathbf{x}(t_i + \Delta t)$).

Tako poenostavljena dvokoračna iteracija velja samo pri uporabi Euler-jeve integracijske metode. Pri uporabi numerično bolj izpopolnjenih postopkov potrebuje integracijska metoda na enem računskem koraku več ovrednotenj odvodov, torej je potrebnih več klicev podprograma **DERIV**.

Če nekoliko razširimo osnovni koncept digitalne simulacije zveznih dinamičnih sistemov, dobimo koncept digitalnih simulacijskih sistemov (orodij, simulacijskih paketov), kot ga prikazuje Sl. 8.4.

Pred simulacijskim tekom se izvedejo določene operacije (npr. inicializacija stanj). Simulacijski tek pa se izvrši s klicem podprograma za integracijo (**INTEG**), ki po potrebi kliče podprogram za izračun odvodov spremenljivk stanja (**DERIV**). V trenutkih, ki jih definira uporabnik, pa integracijski podprogram kliče podprogram (**OUTPUT**), ki skrbi za ustrezno posredovanje rezultatov simulacije. Po izpolnjenem pogoju za končanje simulacijskega teka se izvršijo še morebitne operacije po koncu simulacijskega teka.



Sl. 8.4. Koncept digitalnih simulacijskih sistemov

8.2.1.2 Numerična problematika v simulacijskih orodjih

Običajno uspešno simuliramo dinamične sisteme, ne da bi se sploh zavedali, do kakšnih numeričnih problemov lahko pride. Pri bolj zamotanih problemih pa je pomembno vsaj osnovno poznavanje numerične problematike. Le-to lahko v grobem razdelimo na problematiko numerične integracije in na problematiko algebrajske zanke (Matko in ostali, 1992).

Numerična integracija

Numerična integracija je temeljni simulacijski postopek. Zato ima vsako simulacijsko orodje na voljo večje število integracijskih metod. Na izbiro metode in nekaterih drugih krmilnih parametrov simulacije (npr. dopustna napaka, dolžina računskega koraka,...) vpliva zahteva po *natančnosti*, pogostost *nezveznosti* med simulacijo, *togost* sistema (zelo različne časovne konstante) in *oscilatornost* sistema (poli so blizu imaginarne osi).

Najpogosteje uporabljamo t.i. *enokoračne integracijske metode*, saj so primerne za širok spekter modelov. Za izračun trenutnih stanj (izhodov integratorjev) potrebujejo le eno predhodno vrednost stanj. Zlasti je znana metoda Runge-Kutta-Fehlberg.

Večkoračne metode (npr. Adamsove metode) pa potrebujejo več preteklih stanj. Učinkovitejše so s stališča porabe računalniškega časa, saj zahtevajo manj ovrednotenih odvodov oz. funkcije $f(t, \mathbf{x})$. Neprimerne pa so večkoračne metode, če med simulacijo pogosto nastopajo nezveznosti.

Tako enokoračne kot večkoračne metode pa se delijo še na *eksplicitne in implicitne*. Eksplicitne metode zahtevajo več računalniškega časa, zato pa zagotavljajo boljšo numerično stabilnost.

Razen omenjenih metod so znane t.i. *Gearove metode za toge sisteme* (sistemi z zelo

različnimi časovnimi konstantami). Pri izredno strogih zahtevah glede natančnosti rezultatov pa je smiselno uporabiti ekstrapolacijske metode.

Algebrajska zanka

Pri dobro modeliranem realnem problemu je običajno vse spremenljivke možno izračunati iz spremenljivk stanja (iz izhodov integratorjev). V takem primeru vrstni algoritem lahko razvrsti stavke (bloke) iz simulacijskega programa v pravilen proceduralni vrstni red za izračun odvodov (npr. podprogram DERIV). Vsak problem, v katerem je možno stavke razvrstiti na opisan način, ima lastnost, da je v vsaki zanki simulacijske sheme vsaj en blok z zakasnitvenim atributom (integrator, diskretna zakasnitev, zakasnilni člen,...). Če so modeli v t.i. kanoničnih oblikah, potem ni problemov v razvrščanju.

Zaradi napak v programiranju (nedefinirane konstante, vhodni signali, tipkarske napake v imenih spremenljivk,...), zaradi slabega modeliranja ali pa tudi zaradi narave simulacijskega modela, pa lahko dobimo zanke, v katerih ni blokov z zakasnitvenim atributom. V tem primeru vrstni algoritem ne uspe razvrstiti blokov. Takim zankam pravimo *algebrajske zanke*.

V številnih primerih lahko algebrajske zanke odpravimo z algebraično preureditvijo enačb, ki opisujejo model oz. njegove podmodele. Taka rešitev omogoča znaten prihranek potrebnega računalniškega časa, precej pa se poveča tudi natančnost rezultatov. Zato je preureditev enačb (algebraična eliminacija zanke) vedno prvo, kar je vredno poizkusiti. Nekatera orodja (npr. orodje DYMOLA), ki ne rešujejo le problema simulacije, ampak nudijo tudi močno podporo pri modeliranju, izvršijo ta postopek avtomatsko.

Če pa algebrajskih zank ne uspemo odpraviti z algebraično preureditvijo enačb, imajo številna orodja na voljo t.i. numerični implicitni postopek. Pri tem je za ovrednotenje funkcije $f(t, \mathbf{x})$ potreben iterativni postopek (npr. Newton-Raphsonova metoda), zato je taka simulacija precej dolgotrajna, vprašljiva pa je tudi natančnost rezultatov. Zato je včasih boljša rešitev, da zanko umetno prekinemo s sistemom 1. reda z ojačenjem ena in s kratko časovno konstanto ali pa z diskretno zakasnitvijo. Vendar pa s tem vnesemo novo numerično problematiko - togost.

8.2.1.3 Vrste simulacijskih orodij

Razvoj simulacijskih orodij ima zelo dolgo zgodovino. Analogni računalniki so bili najpomembnejši v šestdesetih in sedemdesetih letih. V šestdesetih letih pa so se začeli razvijati tudi prvi simulacijski paketi za digitalne računalnike. Na razvoj je ključno vplival t.i. standard CSSL. Le-ta je izredno posrečeno definiral strukturne, funkcionalne in sintaktične gradnike bodočih simulacijskih jezikov. Še nekateri današnji najuspešnejši simulacijski jeziki v veliki meri upoštevajo omenjeni standard - npr. simulacijski jezik ACSL.

Simulacijski jeziki CSSL so predstavniki enačbnih simulacijskih jezikov. Njihova sintaksa omogoča opis modelov s pomočjo enačb, podobno kot opisujemo matematične modele. Drugi tipi jezikov so bločni jeziki. Veljajo za manj učinkovite, saj je uporabnik omejen z vgrajenimi bloki (sumatorji, ojačevalni bloki, konstante, prenosne funkcije,...).

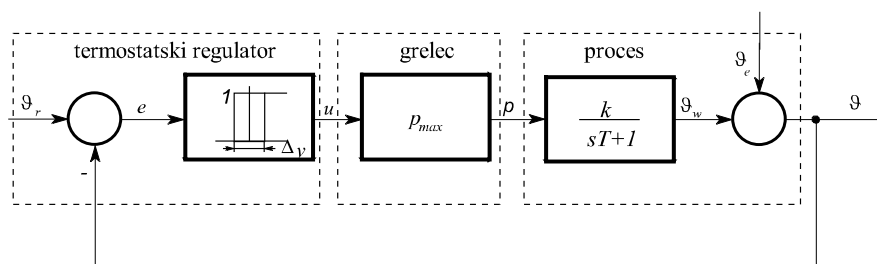
Vse do devetdesetih let so prevladovali enačbni simulacijski jeziki, ki so imeli tekstovni način vnosa modela. V devetdesetih letih, ko so izredno napredovali okenski operacijski sistemi, pa so izdelovalci simulacijske programske opreme začeli izdelovati tudi grafične urejevalnike za vnos simulacijskih modelov (npr. SIMULINK v okolju MATLAB). Taka orodja lahko smatramo kot bločna simulacijska orodja. Uporabnik s pomočjo miške sestavi model tako, da na zaslonu nariše bločno (simulacijsko) shemo. Nekateri grafični urejevalniki so izvedeni kot vmesniki za prevedbo grafičnega vnosa v tekstovni model. Taka orodja omogočajo, da večji uporabniki izkoristijo in kombinirajo dobre lastnosti obeh načinov vnosov. Sicer pa velja, da je tekstovni vnos bolj primeren za izkušene uporabnike, grafični vnos pa za manj izkušene uporabnike, zlasti pa za začetnike.

∇

Primer 8.1: Simulacija ogrevanja prostora

Postopek prevedbe matematičnega v računalniški (simulacijski) model bomo prikazali na problemu regulacije temperature v prostoru.

Za model ogrevanja prostora smo v podpoglavju 7.2.2 (Metode za simulacijo dinamičnih sistemov) razvili simulacijsko shemo (Sl. 7.23). Tokrat bomo simulirali regulacijski sistem z dvopoložajnim stopenjskim (ON/OFF) regulatorjem, ki vključuje in izključuje električno grelo in modelom temperaturnega procesa. Ustrezni bločni diagram prikazuje Sl. 8.5.



Sl. 8.5. Bločni diagram temperaturnega regulacijskega sistema

Enačba

$$\dot{\vartheta}_w + \frac{1}{T} \vartheta_w = \frac{k}{T} p$$

opisuje model procesa v delovni točki. Podatki za simulacijo so naslednji: širina histereze dvopoložajnega regulatorja je $\Delta y = 1 \text{ } ^\circ\text{C}$, moč grelnika je $p = p_{\max} = 5 \text{ kW}$,

temperatura okolice je $\vartheta_e = 15\text{ }^{\circ}\text{C}$, ojačenje procesa je $k = 2\text{ }^{\circ}\text{C/KW}$, časovna konstanta procesa je $T = 1\text{ h}$, začetna temperatura v prostoru je $\vartheta(0)=16\text{ }^{\circ}\text{C}$ oz. $\vartheta_w(0)=1\text{ }^{\circ}\text{C}$. Referenčna temperatura je časovno spremenljiva funkcija in jo opisuje Tabela 8.1..

Tabela 8.1. Želena sobna temperatura

$t[\text{h}]$	0	5.99	6	8.99	9	14.99	15	20.99	21
$\vartheta_r[{}^{\circ}\text{C}]$	15	15	20	20	18	18	20	20	15

Tekstovni vnos modela - simulacija z jezikom po standardu CSSL

Problem bomo simulirali z jezikom SIMCOS (Zupančič, 1992; Zupančič, 1995; Matko in ostali 1992). Z majhnimi spremembami je model možno simulirati s kakršnim koli jezikom tipa CSSL.

Najprej izberemo ustrezna imena, ki jih bomo uporabljali v programu:

ϑ_r	THR	ϑ_e	THE
ϑ	TH	$\vartheta_w(0)$	THW0
$\dot{\vartheta}_w$	THWD	ϑ_w	THW
E	E	p	P
p_{max}	PMAX	Δy	DELTAY
U	U	T	TIMCON
K	GAIN		

Na začetku programa običajno definiramo parametre modela, t.j. konstante in morebitne funkcijske generatorje

"konstante modela

CONSTANT DELTAY=1, PMAX=5, GAIN=2, TIMCON=1

CONSTANT THE=15, THW0=1

"točke funkcijskega generatorja

TABLE REF, 1, 9, ...

0., 5.99, 6., 8.99, 9., 14.99, 15., 20.99, 21., ...
15., 15., 20., 20., 18., 18., 20., 20., 15.

Funkcijski generator definiramo z imenom (REF), s številom neodvisnih spremenljivk (1), s številom točk, v katerih je podana funkcija (9) ter z vrednostmi neodvisne (naslednjih devet podatkov) in odvisne spremenljivke (zadnjih devet podatkov).

Nato je potrebno definirati strukturo modela. Vrstni red stavkov je povsem poljuben.

Referenčni signal v regulacijskem sistemu realiziramo s klicem funkcije za realizacijo funkcijskega generatorja

$$THR=REF(T)$$

kjer je T neodvisna spremenljivka (čas) in REF ime funkcijskega generatorja, ki smo ga definirali s stavkom TABLE.

Pogrešek v regulacijskem sistemu definiramo s stavkom

$$E=THR-TH$$

Vodimo ga v blok, ki modelira stopenjski (ON-OFF) termostatski regulator. Ustrezni model podaja histerezna funkcija s histerezno širino Δy , spodnjim nivojem nič in zgornjim nivojem ena.

CONSTANT STATE =0.

$$U=HSTRSS(E, -\Delta y/2., \Delta y/2., 0., 1., STATE)$$

Zadnji parameter v funkciji HSTRSS je začetni pogoj, ki določa izhodno vrednost v

primeru, če ob prvem klicu pogrešek e leži v področju $-\frac{\Delta y}{2} \leq e \leq \frac{\Delta y}{2}$.

Grelec modeliramo s pomočjo ojačevalnega bloka

$$P=P_{MAX} \cdot U$$

Proces pa simuliramo z indirektno metodo s pomočjo simulacijske sheme, ki jo prikazuje Sl. 7.23. ali pa s pomočjo enačbe 7.72 v podpoglavju Metode za simulacijo dinamičnih sistemov (7.2.2).

$$THWD=-1./TIMCON \cdot THW+GAIN/TIMCON \cdot P$$

$$THW=INTEG(THWD, THW0)$$

Drugi parameter stavka INTEG je začetni pogoj modela v delovni točki. Absolutno temperaturo pa dobimo tako, da temperaturi, ki jo daje model v delovni točki, prištejemo temperaturo delovne točke (t.j. temperaturo okolice).

$$TH=THW+THE$$

Po opisu strukture moramo definirati še krmilne parametre simulacije. Ker želimo opazovati časovne poteke v teku enega dneva, je pogoj za končanje simulacije

CONSTANT TFIN=24

TERMT T.GT.TFIN

Integracija je osrednji postopek vsakega simulacijskega sistema. Dobri simulacijski sistemi imajo več integracijskih postopkov. Če ga posebej ne navedemo, se uporabi

privzeta metoda (običajno Runge-Kutta s prilagodljivim računskim korakom). V modelih, ki vsebujejo histerezo funkcijo, pa je priporočljivo uporabiti metodo s konstantnim računskim korakom. To dosežemo s stavkom

ALGORITHM IALGOR=1, JALGOR=5

Prvi parameter (IALGOR=1) v tem primeru nima pomena, z drugim parametrom (JALGOR=5) pa izberemo metodo Runge-Kutta 4. reda s konstantnim računskim korakom. Komunikacijski interval izberemo s stavkom

CINTERVAL CI=0.02

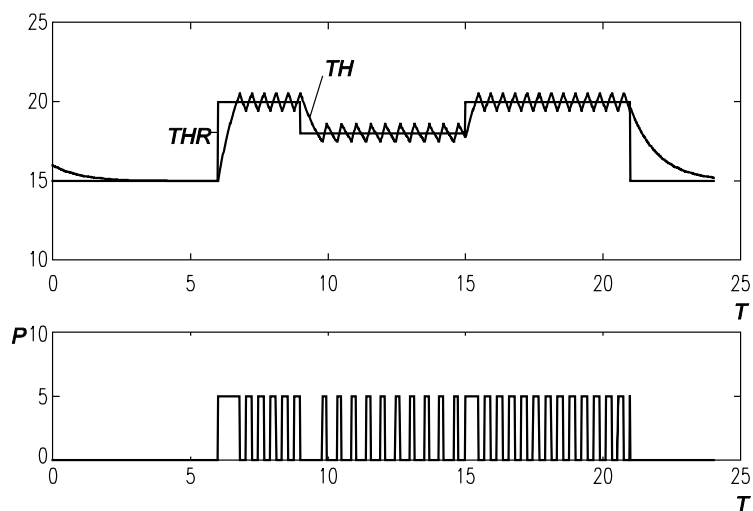
kar pomeni, da ima uporabnik na voljo rezultate simulacije vsake 0.02 h (1.2 min). Ponovno naj poudarimo, da morajo biti enote konsistentne. Čas mora biti povsod izražen v enaki enoti (npr. konstante modela, dolžina simulacijskega teka, komunikacijski interval). Končno definiramo še spremenljivke, ki se med simulacijo izpisujejo na zaslon in v datoteko

OUTPUT 10, THR, P, TH
PREPAR THR, P, TH

Simulacijski program zaključimo s stavkom

END

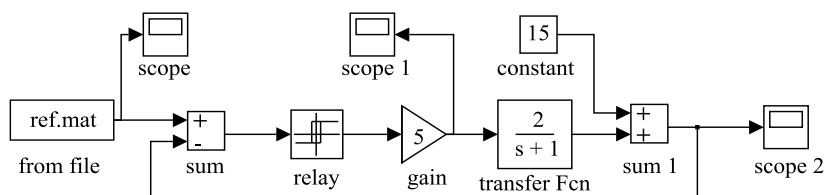
Po procesiranju lahko uporabnik izvede simulacijski tek. Sl. 8.6 prikazuje rezultate simulacije (temperaturo v prostoru ϑ in moč grelca p) v grafični obliki. Grelec se vključuje približno na 30 min, temperatura v prostoru pa niha v pasu 1 °C.



Sl. 8.6. Rezultati simulacije temperaturnega regulacijskega problema

Grafični vnos - simulacija v okolju MATLAB-SIMULINK

S pomočjo orodja MATLAB-SIMULINK na grafični način neposredno vnesemo bločno shemo, ki jo prikazuje Sl. 8.5. Model v okolju MATLAB-SIMULINK prikazuje Sl. 8.7.



Sl. 8.7. Model v okolju MATLAB-SIMULINK

V okno novega modela najprej iz ustreznih knjižnic prenesemo potrebne ikone. Blok `ref.mat` opisuje referenčni signal, bloki `SCOPE` pa prikazujejo rezultate. Ikone s pomočjo miške povežemo, nato pa vsaki ikoni vpišemo ustrezne problemske parametre. Na koncu v posebnem oknu vpišemo krmilne parametre simulacije (vrsto integracijske metode, začetni in končni čas simulacije, itd.). S tem je model pripravljen za eksperimentiranje.

Δ

8.2.2 Računalniško podprto načrtovanje sistemov vodenja

Ena izmed definicij programske opreme, ki jo uporabljamo za analizo in načrtovanje vodenja sistemov - CACSD (angl. Computer Aided Control System Design) oziroma CACE (Computer Aided Control Engineering) se glasi (Denham, 1987):

Programska oprema za CACSD (CACE) je program ali niz zgrajenih programov, ki pokrivajo določeno področje iz sistemske teorije in ki omogočajo uporabniku iz vrst predvidenega kroga uporabnikov aplikacijo konkretne metode na interaktivni način.

Definicija programske opreme za CACSD vsebuje pojem interaktivnega načina uporabe. V Denhamovi definiciji je interaktivni princip v povezavi z že zgrajenimi programi tista nova kvaliteta, ki tovrstno opremo loči od njenih najbližjih predhodnikov, to je programov, ki jih uporabnik sam zgradi na osnovi že pripravljenih knjižnic podprogramov, in katerih izvajanje se po startu opravi brez uporabnikove intervencije (t.i. paketni način oziroma način BATCH). Interakcija je tista lastnost, ki po eni strani omogoča povezavo različnih, že sprogramiranih metod oziroma operacij v celoto, po drugi strani pa enostavno uporabo neke izbrane metode na različnih obravavanih primerih.

8.2.2.1 Orodja za računalniško podprto načrtovanje vodenja sistemov

Po opisu načrtovalnega postopka lahko povzamemo, da se načrtovalec zlasti v fazi modeliranja, analize in sinteze pogosto srečuje:

- z velikim številom različnih metod, s katerimi lahko reši konkretni problem v določeni fazi načrtovalnega postopka;
- z večjim številom različnih algoritmov, ki jih lahko uporabi za že določeno, izbrano metodo (na primer izbira integracijske metode pri simulaciji);
- s kombinacijo več, zelo različnih metod;
- z velikim številom verifikacij in vrednotenj že opravljenega dela postopka.

Če k temu dodamo še nedoločenost sistemov in nezadostno poznavanje ter pomanjkanje informacij o obravnavanih sistemih, kar povzroča težave pri natančni definiciji želenih končnih zahtev, lahko ugotovimo, da je proces načrtovanja v splošnem izredno kompleksen. Zato ga ni možno smotrno in učinkovito obvladovati brez primernih orodij.

Ustrezna programska oprema oziroma paketi CACSD (CACE) omogočajo uporabniku eksperimentiranje, vrednotenje, analizo in sintezo sistemov na enostaven način. Res je sicer, da je potrebno za uspešno uporabo paketov osvojiti določena znanja in si pridobiti nekaj izkušenj. Vendar so napor in potreben čas za to zanemarljivi v primerjavi z lastnim programiranjem, ki zahteva poleg tega še natančno analitično poznavanje metod in kar precej znanja s področja računalništva in numerične analize. Pri uporabi paketov CACSD (CACE) je potrebno poznavanje metod le do takšne mere, da je uporabnik sposoben določiti parametre, ki vplivajo na izvajanje izbranega algoritma, in da je sposoben v nadaljevanju koristno uporabiti dobljene rezultate. Na ta način omogočajo paketi CACSD (CACE) nekajkrat uspešnejšo uveljavitev poznavanja in razumevanja sistemske teorije ter uporabnikovih izkušenj in intuicije. V odvisnosti od znanja, dostopnih podatkov in kompleksnosti, je možno tako rešiti problem v nekaj urah ali dneh z minimalnimi znanji računalništva in manjšim poznavanjem uporabljenih metod. Zadnje je še zlasti pomembno pri sodobnih metodah iz sistemske teorije, ki ponujajo možnost hitrejše, stabilnejše in optimalnejše regulacije, a so analitično in numerično veliko bolj zapletene.

Reševanje problemov, kjer je računalnik izrazito slabši, pa je potrebno prepustiti načrtovalcu. Človeške prednosti so zlasti:

- zmožnost abstrakcije, poenostavljanja in posploševanja;
- zmožnost obvladovanja nepopolnih in slabše definiranih problemov;
- izkušnje, intuicija in sposobnost logičnega sklepanja;
- prilagodljivost in fleksibilnost;
- sposobnost razpoznavanja in povezovanja vzorcev.

Za namen paketov velja v celoti Sneidermanova definicija (Sneiderman, 1987), ki pravi,

da je cilj vseh interaktivnih sistemov poenostavitev dela. To pomeni izločanje človeških akcij v fazah, ko ni potrebna nobena sodba, ne induktivna in ne deduktivna. Na ta način se rešimo utrudljivih del in potencialnih napak, ki iz njih ponavadi izvirajo. Pridobimo pa možnost, da se bolj skoncentriramo na kritične odločitve, ustrezno planiranje ter na reševanje nepredvidljivih situacij.

Pomen paketov CACSD (CACE) se izredno poveča v povezavi z dvema pomembnima področjema, to je z industrijsko prakso in izobraževalnim procesom.

Znano je, da temelji industrijska praksa pretežno na pridobljenih izkušnjah. Dogaja se celo, da inženirji ne vedo, zakaj se odločijo ali predlagajo konkretno rešitev oziroma določeno regulacijsko tehniko. Argumenti so v tem primeru "prej je delovalo" ali "vedno do sedaj je bilo tako". Obstajajo pa tudi okolja, v katerih bi sicer radi uporabili nove metode, a si jih ne upajo, ker je navidezna kompleksnost problemov, v primerjavi s klasičnimi pristopi, ki so dobro vpeljeni in preizkušeni, prevelika. Paketi CACSD (CACE) z vgrajenimi simulacijskimi sposobnostmi lahko te razmere močno spremenijo, saj ponujajo učinkovit način pridobivanja praktičnih izkušenj za različne regulacijske tehnike in metode. Enostavni način preizkušanj in preverjanj lahko mnogo pripomore, da bi v industriji dobili zaupanje v "nove in nepreverjene" metode. Obenem pa je to tudi način, ki omogoča izobraževanje oziroma vzgojo ustreznega profila inženirjev.

Podobne ugotovitve veljajo tudi za redne izobraževalne procese, kjer ponuja interaktivno učenje smotrni način izobraževanja in usposabljanja kvalitetnih, učinkovitih in inovativnih inženirjev. Zato je že Walker s sodelavci (Walker in ostali, 1984), kot vzporedni cilj paketov CACSD (CACE), definiral tudi:

- pospeševanje in vplivanje na nove ideje, produkte in pristope;
- hitro preobrazbo v kreativnega inženirja za vodenje.

Po Schmidu (Schmid, 1982) je vzporedni cilj paketov CACSD (CACE) tudi, da podpirajo in privzgaajo sistematičnost v načrtovalnih postopkih in strategijah.

Na razvoj programske opreme za CACSD sta odločilno vplivala razvoj teorije vodenja in razvoj računalnikov ter spremljajočih računalniških znanj. Začetek razvoja sega v konec šestdesetih let dvajsetega stoletja in poteka od enostavnih programov za zelo specifične in enkratne izvedbe, preko programskih paketov, ki so jih razvijali specialisti s področja vodenja v akademskem okolju, do profesionalnih izdelkov, kjer so imeli poleg računalniških programerjev pomembno besedo tudi numerični analitiki.

Sodobni programski izdelki vključujejo robustne numerične postopke iz knjižnic kot so EISPACK, LINPACK, NAG idr. ter sodobne pristope kot so objektno orientirano programiranje in visoko kvalitetni in interaktivni grafični vmesniki.

Med komercialnimi paketi na področju CACSD najbolj izstopata imeni MATLAB in MATRIX_x. Oba paketa sta se razvila iz prve verzije paketa MATLAB, ki je nastala leta 1980 in je podpirala operacije linearne algebre v matričnem okolju.

Glavni konkurent omenjeni trojici pa je programski paket Mathematica, ki temelji na simboličnem računanju, vendar omogoča tudi numerične izračune. Njegovi začetki izhajajo iz matematičnega in ne inženirskega okolja.

Iz poteka razvoja programskih paketov CACSD vidimo, da je imel programski paket MATLAB zelo veliko, če ne najbolj opazno vlogo. Zaradi tega ga bomo nekoliko podrobneje predstavili ter ga uporabili tudi pri obravnavi primera, ki naj bi ilustriral nekatere možnosti programskih paketov za računalniško podprto načrtovanje vodenja sistemov.

Trendi razvoja programskih paketov CACSD so usmerjeni tako v strukturne kot vsebinske izboljšave. Strukturno se najpogosteje pojavljata pojma paralelizacije in objektne orientiranosti. Težišče dogajanja se iz numerično naravnanih paketov seli proti simboličnim, ki pa so prav tako zmogljivi tudi za numerične operacije. Vsebinsko je dominantna usmerjenost v razvoj paketov, ki omogočajo analizo in načrtovanje hibridnih sistemov. To so tisti sistemi, pri katerih najdemo tako zvezne (npr. zvezni kemijski procesi) kot diskretne dogodke (npr. izdelčna proizvodnja). Novosti na področju CACSD najdemo opisane v (IEEE, 1995).

8.2.2.2 Programski paket MATLAB

MATLAB (MATrični LABoratorij) je interaktivno programsko orodje za numerično reševanje problemov. Razvijati so ga začeli v univerzitetnem okolju in je šele kasneje postal komercialni produkt. Njegov osnovni namen je bilo omogočiti enostaven dostop do programskih knjižnic linearne algebre znanih pod imenom LINPACK in EISPACK. MATLAB je bil sicer najprej razvit za reševanje problemov s področja linearne algebre, vendar se je kaj hitro izkazal kot zelo primeren za pakete CACSD zaradi naslednjih lastnosti (Matko et al., 1993):

- zelo sposobne in fleksibilne izrazne moči interaktivnega vmesnika, ki temelji na ukaznem načinu dialoga;
- zelo podobne podatkovne strukture (matrike), kot je potrebna pri zapisu sistema v prostoru stanj;
- uporabljenih robustnih numeričnih algoritmov (EISPACK in LINPACK);
- uporabe osnovnih principov neposrednega upravljanja (angl. direct manipulation) za matrične operacije.

MATLAB je imel močan vpliv na razvijalce drugih sorodnih programskih orodij.

Poleg že omenjenih, ima MATLAB še eno odliko, in to je enostavna razširljivost s podprogrami, ki jih glede na podaljšek imen imenujemo M datoteke. To je bil vzrok, da so nastali programski moduli (angl. Toolbox) - zbirke podprogramov programskega orodja MATLAB, ki so namenjeni različni namenski uporabi.

Naš namen ni podrobna predstavitev elementov paketa MATLAB, zato bomo naredili le

kratek pregled paketa.

MATLAB pozna v bistvu le eno osnovno vrsto objekta - pravokotno numerično matriko s kompleksnimi elementi. Vse spremenljivke so torej matrike. V posebnih primerih je matrika 1×1 razumljena kot skalar in stolpična ali vrstična matrika kot vektor. Matrike lahko vstavimo v MATLAB na več načinov:

- kot zaporedje posameznih elementov;
- z vgrajenimi izrazi ali funkcijami;
- z M datotekami;
- iz zunanjih podatkovnih datotek.

V MATLAB-u imamo vgrajene razne matrične operacije, ki pa jih lahko izvajamo tudi med skalarji in v glavnem tudi mešano med matrikami in skalarji.

MATLAB je interpreterski jezik. Vsak izraz, ki ga odtipkamo je tolmačen (interpretiran) in ovrednoten posebej. Vnosi v MATLAB-u so ponavadi oblike

spremenljivka = izraz

ali kar

izraz

Izrazi so sestavljeni iz operatorjev, funkcij in imen spremenljivk. Izraz se ovrednoti in nastane matrika, ki se prikaže na zaslonu in ji je določena spremenljivka, ki jo nato lahko uporabljamo. Če ne navedemo imena spremenljivke in prireditvenega enačaja, priredi MATLAB izraz spremenljivki *ans*, ki jo lahko kasneje uporabimo.

Funkcije vgrajene v MATLAB delimo na skalarne, vektorske in matrične, glede na spremenljivke s katerimi delujejo. Da bi lahko izraze med seboj lažje povezovali v podprograme, ima krmilne stavke (*for*, *while*, *if*) in relacijske operatorje. Podprogrami so sekvence ukazov zapisane v datoteko, ki ima pri svojem imenu podaljšek *.m*. Ločimo dva tipa M datotek: funkcijske datoteke (delujejo kot funkcija) in datoteke, ki vsebujejo samo sekvence ukazov oziroma vpisov.

MATLAB ima močno grafično podporo. Numerične rezultate lahko prikažemo v dvo ali trodimenzionalnih grafih najrazličnejših izvedb in jih poljubno označimo.

Z možnostjo, da lahko uporabnik tudi sam razvija funkcije, ki so zanimive le za določeno vrsto uporabe, so bili, kot smo že omenili, razviti programski moduli za posamezne veje tehnike in naravoslovja. Tako poznamo programske module za obdelavo signalov, načrtovanje sistemov vodenja, robustno vodenje, identifikacijo sistemov, optimizacijo, kemometrijo, μ analizo in sintezo, identifikacijo v prostoru stanj, mehko logiko, nevronske mreže in mnoge druge.

Nekoliko drugače od preostalih programskih modulov je zastavljen *Simulink*. Simulink je programski modul za simulacijo, ki v osnovnem MATLAB-u in ostalih programskih

modulih ni omogočena na uporabniško prijazen način. V Simulinku grafično z bloki, ki predstavljajo dinamične podsisteme, sestavimo simulacijsko shemo linearnega, nelinearnega, zveznega, diskretnega ali hibridnega sistema in ga nato simuliramo z izbrano metodo. Simulink služi kot zelo priročno dopolnilo ostalim modulom.

Delo s programskim paketom MATLAB in v njegovem okviru z modulom za načrtovanje vodenja ter Simulinkom ilustrirajmo z naslednjim primerom.

▽

Primer 8.2: *Uporaba programskega paketa MATLAB za načrtovanje sistema vodenja*

Oglejmo si primer načrtovanja sistema vodenja z MATLAB-om. Prikazali ga bomo z načrtovanjem vodenja hidravličnega sistema, ki je sestavljen iz treh povezanih shranjevalnikov tekočine. Postopek modeliranja in linearizacije je opisan v podpoglavju o matematičnem modeliranju procesov. Model lahko predstavimo v različnih oblikah, kot so v navadi pri modeliranju dinamičnih sistemov. V našem primeru bomo vnesli proces v obliki prenosne funkcije. Linearizirani model opisujeta prenosni funkciji (enačbi (7.43) in (7.36)), podpoglavje 7.2.1 – Modeliranje procesov)

$$G_{P1}(s) = \frac{H_3(s)}{H_1(s)} = \frac{R_3}{R_1(R_2C_2s + 1)(R_3C_3s + 1)}$$

$$G_{P2}(s) = \frac{H_1(s)}{\Phi_I(s)} = \frac{R_1}{R_1C_1s + 1}$$

pri čemer so konstante $R_1=0.5$, $R_2=1$, $R_3=1$, $C_1=10$, $C_2=10$, $C_3=10$.

Prikazali bomo primer za postopek načrtovanja PID regulacije z nastavitvenimi pravili ob uporabi Bodejevega diagrama, ki je razložen v podpoglavju o načrtovanju proporcionalno-integrirno-diferenčne regulacije (7.3.3.3).

Ogledali si bomo samo primer načrtovanja sledilnega vodenja, brez upoštevanja vhoda za motnjo. To pomeni, da lahko prenosni funkciji združimo v eno.

$$G_P(s) = G_{P1}(s)G_{P2}(s) = \frac{R_3}{(R_1C_1s + 1)(R_2C_2s + 1)(R_3C_3s + 1)}$$

oziroma, če vnesemo podatke in zmnožimo posamezne člene

$$G_P(s) = \frac{1}{500s^2 + 200s + 1}$$

Prenosne funkcije vnašamo v polinomski obliki, kot dva vektorja, ki vsebujeta koeficiente števca oziroma imenovalca prenosne funkcije. Ime spremenljivk bomo izbirali tako, da bo njihov pomen bralcu čim bolj nazoren.

```
stevec=[0 0 0 1];
```

```
imenovalec=[500 200 25 1];
```

Izberemo si zanimivo frekvenčno področje; v našem primeru je to 200 točk na frekvenčnem področju od 0.01 rad/min do 1 rad/min z logaritemskim razmikom, kot je v navadi za frekvenčne odzive.

```
w=logspace(-2,0,200);
```

Frekvenčni odziv procesa (amplitudni in fazni Bodejev diagram) dobimo s komando `bode`, ki diagram tudi izriše.

```
bode(stevec,imenovalec,w);
```

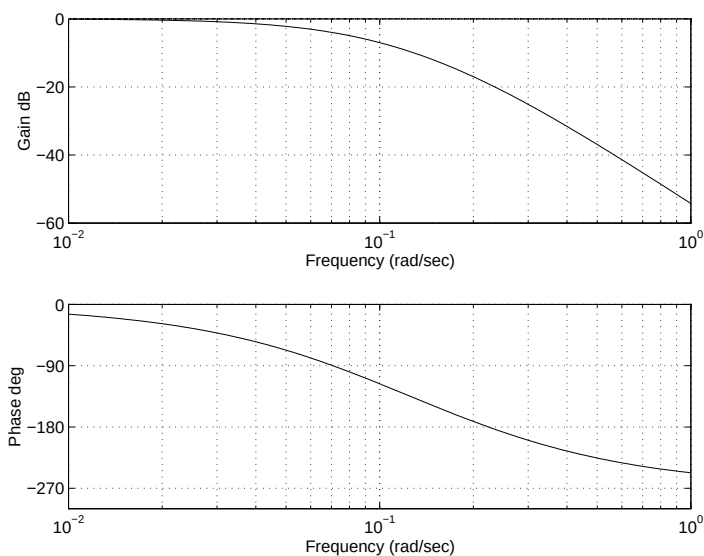
Diagrama prikazuje Sl. 8.8. Enote na abcisi so v našem primeru rad/min in ne rad/sec kot je avtomatsko označil MATLAB, kar pa bi lahko z dodatnimi ukazi spremenili.

Parametre P, PI in PID regulatorja izračunamo z nastavitvenimi pravili, ki smo jih predstavili v poglavju o proporcionalno-integrirno-diferencirni regulaciji (7.3.3.3).

Za izračun P regulatorja potrebujemo absolutno vrednost frekvenčne karakteristike pri faznem kotu -162° . Frekvenco, pri kateri ima fazni odziv želeno vrednost, dobimo tako, da poiščemo tisto vrednost frekvence, pri kateri bo absolutna vrednost vsote faznega odziva in vrednosti 162 najmanjša.

```
[faza162,i]=min(abs(faza+162));
```

```
w162=w(i)
```



Sl. 8.8. Amplitudni in fazni Bodejev diagram hidravličnega procesa

Ojačenje proporcionalnega regulatorja je inverzna vrednosti ojačenja pri izračunani frekvenci.

```
[amp162, faza162]=bode(stevec, imenovalec, w162);  
Kp=1/amp162;
```

Za izračun parametrov PI in PID regulatorja pa potrebujemo frekvenco in ojačenje v točki, kjer fazni kot frekvenčnega odziva modela procesa doseže -144° .

```
[faza144, j]=min(abs(faza+144));  
w144=w(j)
```

Iz teh podatkov izračunamo vrednosti parametrov na naslednji način:

```
[amp144, faza144]=bode(stevec, imenovalec, w144);  
Kp=1/amp144;  
Ti=2*pi/w162;  
Td=0.1*Ti;  
T1=Td/10;
```

Uporabljena nastavitvena pravila temeljijo na tem, da zagotovijo potrebni fazni razložek. Zato bomo načrtovane regulatorje vrednotili tako, da si bomo ogledali razložke regulacijskih sistemov, ki jih zagotavljajo ti regulatorji. Najprej moramo izračunati števec in imenovalec posameznih regulatorjev.

```
st_p=Kp;  
im_p=1;
```

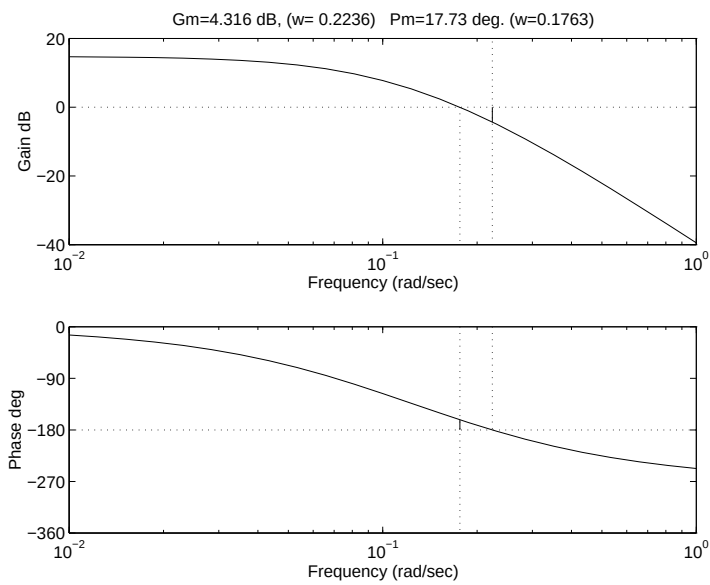
```
st_pi=Kp*[Ti 1];  
im_pi=[Ti 0];
```

```
st_pid=Kp*[Ti*Td+Ti*T1 Ti+T1 1];  
im_pid=[Ti*T1 Ti 0];
```

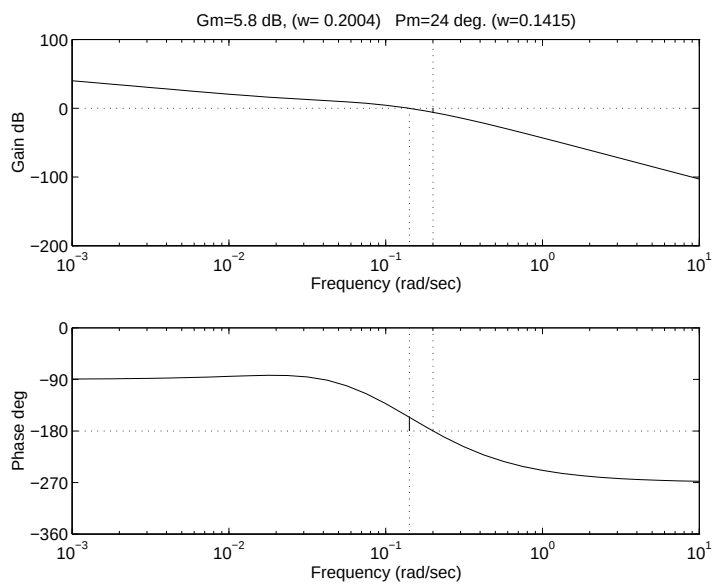
Ojačevalni in fazni razložek celotnega sistema vodenja, ki pomenita njegovo mero robustnosti dobimo z ukazom `margin`. Da bi lahko določili razložka, moramo izračunati odprtozanko prenosno funkcijo. To dobimo tako, da zmnožimo števca in imenovalca procesa in regulatorja, kar dosežemo s konvolucijo. Rezultat je grafični in numerični prikaz rezultatov.

```
margin(conv(stevec, st_p), conv(imenovalec, im_p));  
margin(conv(stevec, st_pi), conv(imenovalec, im_pi));  
margin(conv(stevec, st_pid), conv(imenovalec, im_pid));
```

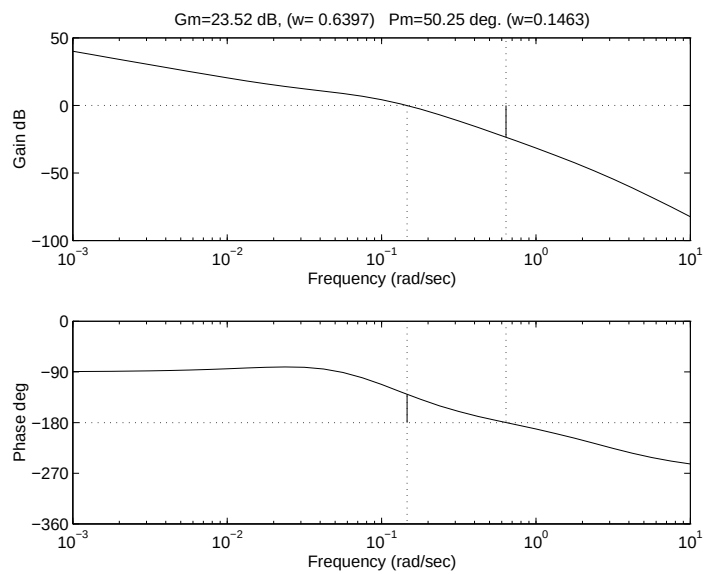
Rezultate vrednotenja prikazujejo Sl. 8.9, Sl. 8.10 in Sl. 8.11.



Sl. 8.9. Amplitudni in fazni Bodejev diagram sistema s P regulatorjem z vrisanimi in izračunanimi razločki

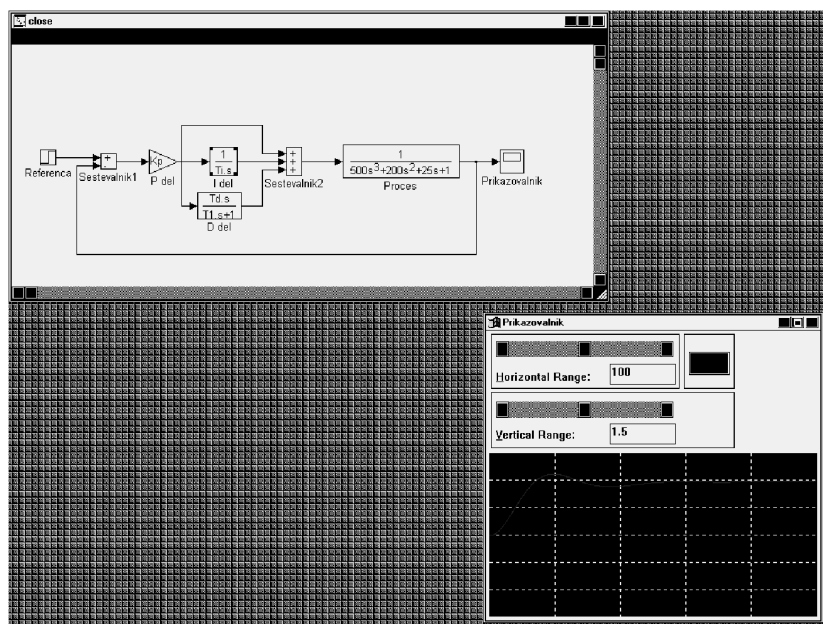


Sl. 8.10. Amplitudni in fazni Bodejev diagram sistema s PI regulatorjem z vrisanimi in izračunanimi razločki



Sl. 8.11. Amplitudni in fazni Bodejev diagram sistema s PID regulatorjem z vrisanimi in izračunanimi razločki

Iz izračunanih razločkov vidimo, da ima P regulator najslabšo robustnost (tako kot predvideva pristop je razloček okoli 18°). Najboljšo karakteristiko pa ima zaprtozančni sistem vodenja s PID regulatorjem.



Sl. 8.12. Simulacijska shema in odziv zaprtozančnega sistema s PID regulatorjem

Zaprtozančni sistem z izračunanim regulatorjem bi lahko simulirali v samem MATLAB-u, vendar je SIMULINK priročnejše orodje. Na Sl. 8.12 sta prikazana simulacijska shema zaprtozančnega sistema s PID regulatorjem in odziv sistema na stopničasto spremembo referenčne vrednosti.

Δ

8.3 Orodja za sistemsko analizo in razvoj programske opreme

Pričujoče podpoglavje je namenjeno predstavitvi nekaterih orodij, ki jih uporabljamo v okviru realizacije celotnega življenjskega cikla programske opreme. Seveda je takih orodij zelo veliko, zato podrobna predstavitev posameznega orodja ne bi imela smisla. Zaradi tega se bomo omejili bolj na predstavitev osnovnih principov in zahtev, ki naj bi jih orodja izpolnjevala, manj pa na lastnosti posameznih predstavnikov.

8.3.1 Orodja za računalniško podprto programsko in sistemsko inženirstvo

Računalniško podprta orodja sistemskega in programskega inženirstva - orodja CASE - avtomatizirajo metode in metodologije sistemskega in programskega inženirstva ter zagotavljajo prijazno in razumljivo povezavo z uporabniki. Podpirajo analizo problemske domene, specificiranje, razvoj in vzdrževanje programske opreme, povratno inženirstvo ter aktivnosti upravljanja in vodenja projektov programske opreme.

8.3.1.1 Splošno o orodjih CASE

Kaj je CASE

Precej bolj kot slovenski prevod Računalniško podprto programsko in sistemsko inženirstvo je med strokovnjaki s področja računalništva poznana originalna kratica CASE. Kratica CASE je bila sprva tolmačena kot Computer Aided Software Engineering (računalniško podprto programsko inženirstvo), ker se je to inženirsko področje ukvarjalo predvsem z razvojem in implementacijo računalniških programov. Z rastočo zahtevnostjo le teh in z razvojem informacijske tehnologije se je vse bolj uveljavljal sistemski pogled na to področje, ki je zajel v obravnavo celotne računalniške sisteme, vpete v širše okolje. Sistemski pogled je bistveni element *sistemskega inženirstva*, ki smo ga podrobneje predstavili v drugem poglavju tega dela. V tem smislu so CASE začeli tolmačiti kot Computer Aided Systems Engineering (računalniško podprto sistemsko inženirstvo).

V zadnjem času, ko informacijska tehnologija omogoča avtomatizacijo raznovrstnih procesov, je kratica CASE dobila najbolj celosten pomen, to je Computer Automated

Systems Engineering (računalniško avtomatizirano sistemsko inženirstvo). Slednja je zato tudi najprimernejša, kadar jo uporabljamo v zvezi z računalniško podprtimi sistemi vodenja procesov. Ti sistemi so največkrat t.i. sistemi realnega časa (Real-Time Systems), kjer programska oprema predstavlja najpomembnejšo komponento, s katero so realizirani koncepti vodenja. S tega stališča je poznavanje in uporaba CASE nujna tudi za doseganje kakovosti računalniško podprtih sistemov vodenja v vsem njihovem življenjskem ciklu.

Računalniško avtomatizirano programsko in sistemsko inženirstvo vedno gledamo v povezavi z drugimi inženirskimi panogami, kot je npr. inženirstvo vodenja sistemov. Ker je na ta način vpeto v okolje, se je udomačil izraz *Okolje računalniško avtomatiziranega programskega in sistema inženirstva* (CASE Environment), v katerega so zajeti vsi bistveni sestavni deli. To so metode in metodologije, računalniška orodja in ljudje - uporabniki. Osnovna filozofija enotnosti okolja temelji na skladni povezanosti vseh treh elementov, ki se jo lahko doseže na več načinov, odvisno od posameznega elementa. Primeri tega so: enaka semantika in sintaksa modelov, skupna podatkovna baza ali sistem povezav med posameznimi orodji (enotna arhitektura), timsko delo, itd.

Kaj se pričakuje od orodij CASE

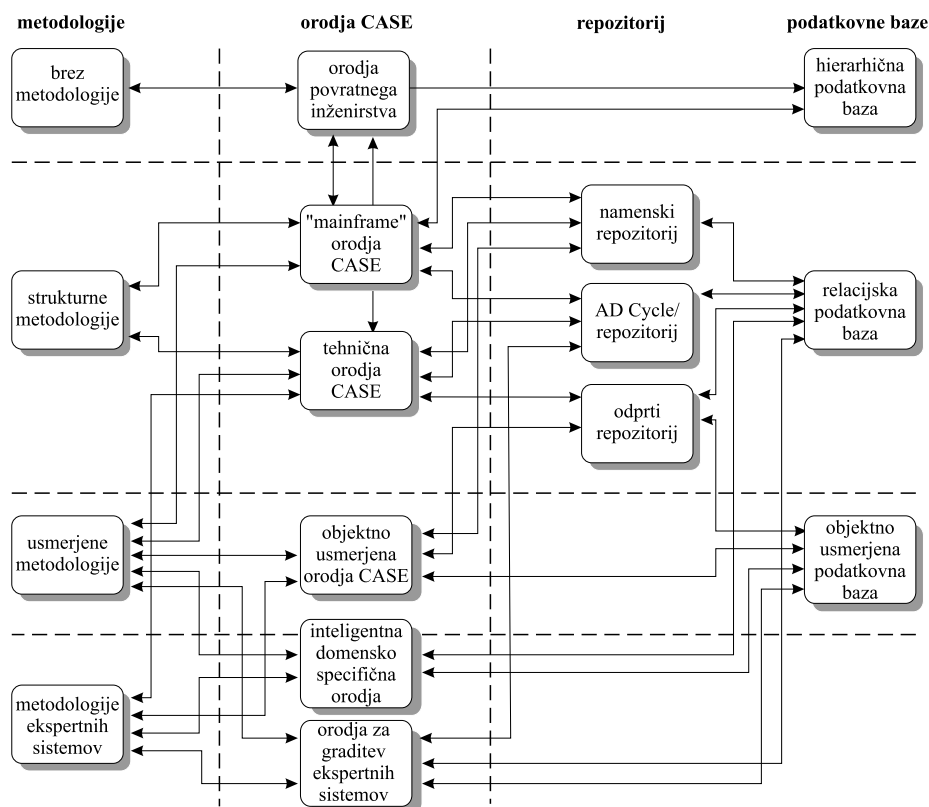
Od orodij CASE se v prvi vrsti pričakuje, da v čimvečji meri avtomatizirajo razvojne faze sistemov. To je možno doseči z uporabo tehnik dekompozicije, s katerimi se dekomponira sistem na obvladljive in skladne sestavne dele. Pri tem orodja uporabljajo grafične polformalne modele, kot so raznovrstni diagrami, ki smo jih predstavili v okviru podpoglavja o programskem inženirstvu v podpoglavju 7.5.1. Prav tako orodja podpirajo tudi ostale faze življenjskega cikla sistemov. Končni cilj orodij CASE je avtomatska generacija izvršljive kode iz specifikacij. K temu cilju vodijo delni cilji, ki prikazujejo koristi uporabe orodij CASE v praksi (Martin, 1991). Istočasno so to tudi zahteve, ki jih mora izpolnjevati vsako moderno orodje CASE. To so:

- povečanje produktivnosti in izboljšanje kakovosti rešitve (programske opreme ali sistema);
- zagotovitev ponovljive kakovosti rešitev;
- zmanjšanje cene rešitve, predvsem pa vzdrževanja;
- pospešitev procesa razvoja rešitev;
- povečanje sinergijskega učinka zaradi uporabe vrste metod in metodologij;
- poenotenje dela projektnih skupin;
- izboljšanje komunikacije v projektnih skupinah;
- izboljšanje dokumentacije.

Naštete zahteve podpirajo orodja CASE z vgrajenimi koncepti skladnosti, completeness ter standardnosti modelov, kode in dokumentov.

Pregled zgodovine orodij CASE

Zgodovinski razvoj orodij CASE je tesno povezan z razvojem metodologij za razvoj programske opreme in informacijske tehnologije v splošnem, s katerima so orodja CASE šla skozi postopne faze razvoja (Harmon in Hall, 1993) (Sl. 8.13). V prvo razvojno fazo spadajo orodja CASE prve generacije. Orodja CASE, ki so danes najbolj razširjena, to so orodja druge in tretje generacije, spadajo v drugo razvojno fazo, orodja CASE, ki bodo prevladovala v bodočnosti, pa spadajo v tretjo razvojno fazo. Pri tem je potrebno povedati, da so metodologije zadnje razvojne faze že precej dozorele in se vedno več uporabljajo.



Sl. 8.13. Vzporeden razvoj metodologij, orodij CASE in informacijske tehnologije

Prva generacija orodij CASE se je pojavila v začetku sedemdesetih let. Ta orodja so bila namenjena za delo na "mainframe" računalnikih. Njihova značilnost je bila njihova neprenosljivost in usmerjenost podpora poslovnim rešitvam na osnovi programskega jezika COBOL.

Druga generacija orodij CASE sega v začetna osemdeseta leta. Nudila so podporo za risanje modelov, ki jih zahtevajo strukturne metode analize, načrtovanja in programiranja. Dalje so podpirala preverjanje pravil posameznih metodologij in imela so podatkovni slovar. Uporabniški vmesnik je bil močno odvisen od proizvajalca,

uporabniška prijaznost je bila slaba, grafika skromna, možnost kakovosti reprodukcije dokumentacije po želji uporabnika pa precej omejena. Slepo sledenje strukturnim metodam brez globljega razmisleka o vsebini problemov, različno razumevanje in interpretiranje vpeljanih zakonitosti, je povzročilo čezmerno kopičenje redundantne dokumentacije.

Tretja generacija orodij CASE se je pojavila v poznih osemdesetih letih. Zanje so značilni takoimenovani repozitoriji, ki so dejansko urejene zbirke vseh relevantnih podatkov o vsebini in izvajanju projektov programske opreme. Orodja so postala odprta, tako da se je možnost združevanja z drugimi orodji precej povečala. Uporabniški vmesniki so bili že precej poenoteni tako na delovnih postajah, "mainframe" računalnikih kot tudi na osebni računalnikih.

V začetku devetdesetih let je bilo opaziti zastoj pri širjenju uporabe orodij CASE, predvsem pa velik zastoj v trženju. Proizvajalci programske opreme preprosto niso hoteli tvegati, da bi njihove rešitve postale v nekaj letih neuporabne. Najočitnejši razlogi za takšno stanje so bili:

- še vedno ni bilo dovolj zadovoljivih povezav med orodji CASE in drugimi programskimi orodji;
- kasnitev IBM-ovega "A/D cycle-a", ki naj bi postal standard za povezljivost rešitev in programskih orodij;
- negotovost, kaj bodo prinesle nove tehnologije, kot npr. strežnik-odjemalec, objektna usmeritev, itd.;
- neučinkovitost strukturnih metod in nezrelost objektno usmerjenih metod v življenjskem ciklu sistemov, predvsem sistemov realnega časa.

Orodja CASE danes

Uporaba orodij CASE se je v današnjem času spet razširila, saj so bili odpravljeni vsi glavni zadržki, ki so zavirali razvoj v začetku devetdesetih let. Poudarek, ki so ga takrat imela orodja na podpori množice zelo različnih metod, se je sedaj premaknil na podporo integracije metod ali podporo integriranim metodam, ki obravnavajo problematiko z vidika zelo različnih domen.

Na področju programskega inženirstva se pojavlja vse več standardov tako od proizvajalcev kot tudi nacionalnih in mednarodnih strokovnih združenj. Najpomembnejši standardi, ki neposredno zadevajo orodja CASE, so:

- ISO/IEC 14102: Information technology - Guideline for the evaluation and selection of CASE tools;
- IEEE 1348: Recommended practice for the adoption of CASE tools;
- IEEE P1209: CASE Tool Evaluation Standard;
- IEEE 1175/D11: Reference model for Computing System Interconnection;

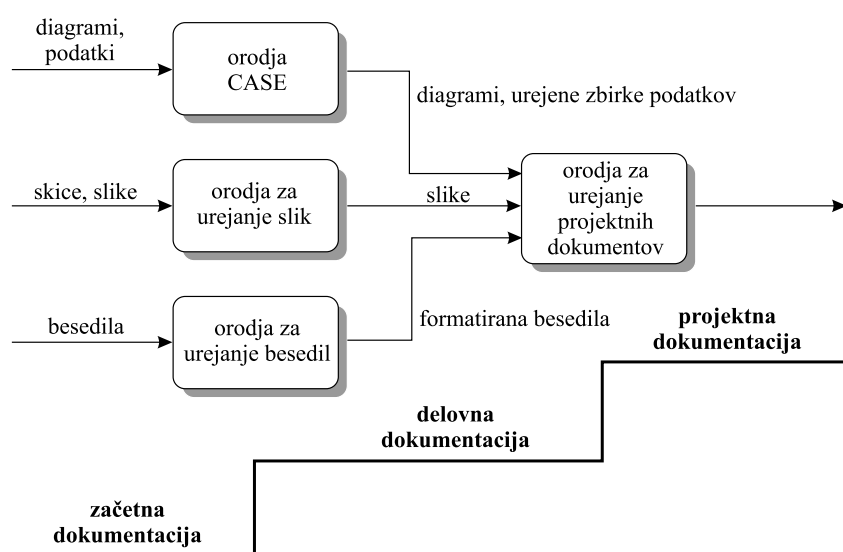
- ISO/IEC 12207: Information technology - Software life-cycle processes;
- ISO/IEC 9126: Information technology - Software product evaluation.

Prisotnost standardov in vse večje zmožnosti računalniške tehnologije vplivajo na rast uporabe orodij CASE. Glede na namen se klasična razdelitev orodij CASE na

- višja orodja CASE ("upper CASE" ali tudi "front-end CASE"), ki podpirajo fazi analize in načrtovanja in
- nižja orodja CASE ("lower CASE" ali tudi "back-end CASE"), ki podpirajo generacijo programske kode in testiranje

vse bolj spreminja v delitev glede na končni proizvod, to je na orodja, ki imajo za končni proizvod dokumentacijo, in na prototipna orodja, kjer je končni rezultat delujoč prototip (Microgold, 1998). V obeh primerih je orodje integrirano, kar pomeni, da združuje različne tipe namenskih programskih orodij.

V prvem primeru (Sl. 8.14) združuje namenska orodja za:



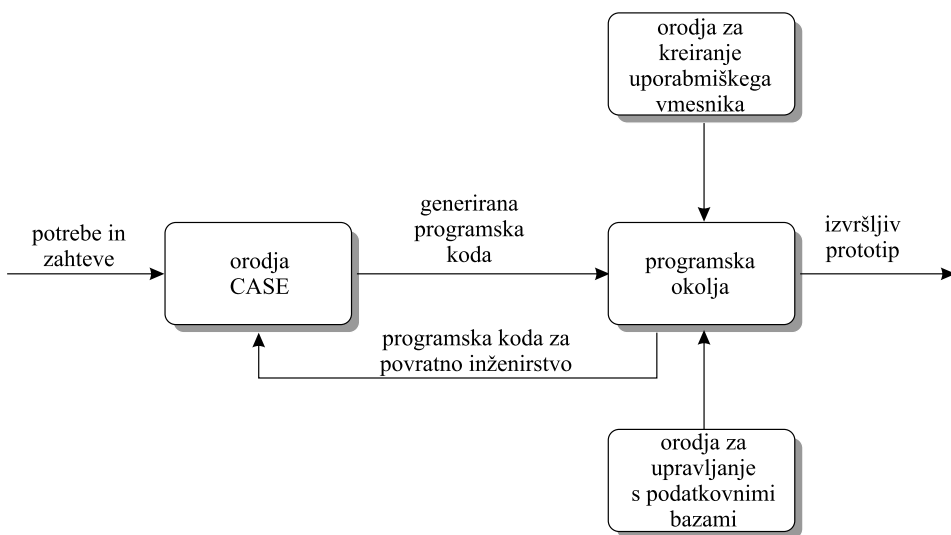
Sl. 8.14. Komponente orodja CASE za generiranje dokumentacije in proces razvoja

- podporo metod programskega ali sistemskega inženirstva (risanje polformalnih modelov oziroma diagramov v skladu s pravili izbrane metode, preverjanje skladnosti in kompletnosti);
- risanje zaslonov (uporabniškega vmesnika) in ostalih grafičnih predstavitev, ki jih prvo namensko orodje ne podpira;
- urejanje besedil, ki so lahko preprosta ali pa je njihova struktura določena po posebnih pravilih;

- podporo projektnih aktivnosti in večuporabiškemu načinu dela.

Na sliki je prikazan tudi proces nastajanja končnega proizvoda, to je dokumentacije, ki pa je seveda za različne aktivnosti različnih faz življenjskega cikla različna.

V drugem primeru (Sl. 8.15) integrirano orodje CASE združuje orodje za podporo metod programskega ali systemskega inženirstva in vsa ostala orodja, potrebna za generacijo delujočega prototipa. Proces razvoja se v tem primeru konča v fazi izvedbe.



Sl. 8.15. Komponente orodja CASE za generiranje delujočega prototipa in proces razvoja

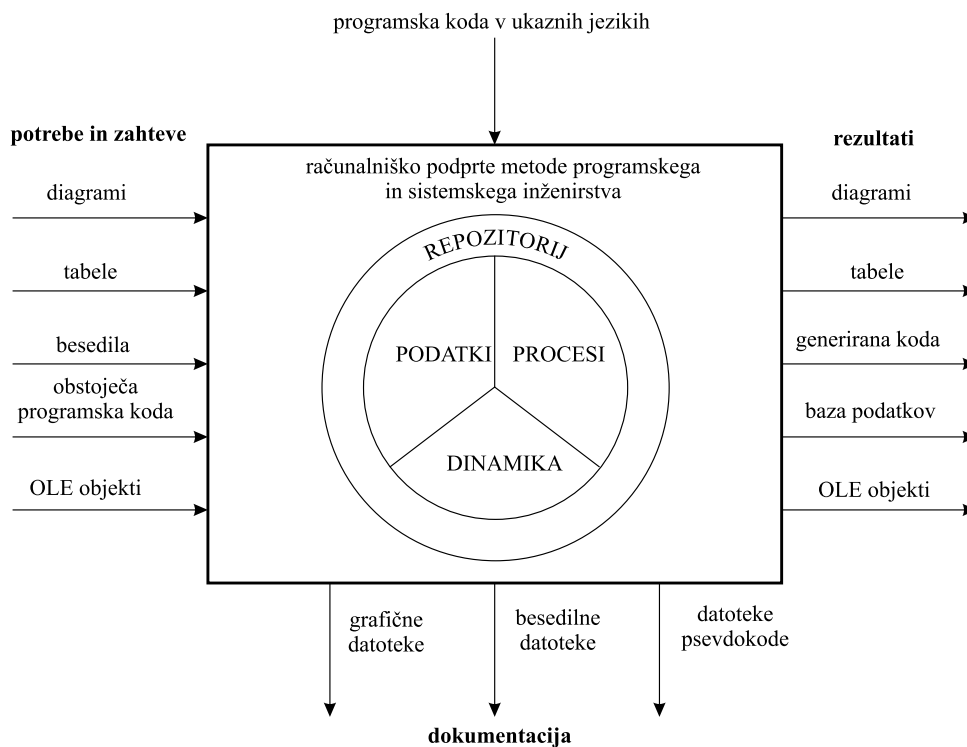
Današnja orodja CASE se od klasičnih razlikujejo ne samo po združitvi različnih funkcij v celoto, ampak tudi po strukturi in povezavi z okoljem. Osrednji del modernega orodja CASE predstavlja repozitorij. To je v bistvu razširjeni klasični podatkovni slovar, ki poleg formaliziranega popisa podatkov, njihovih karakteristik, pregleda uporabe in strukture vseh polformalnih modelov in besedil, vsebuje še informacije o procesu izvajanja projekta. Pregled vseh vhodov v sodobno orodje CASE in generiranih izhodov, je na razumljiv način podan na Sl. 8.16.

Vsa moderna orodja CASE, kot osnovni pripomočki za izdelavo in vzdrževanje polformalnih modelov, omogočajo delo z modeli, ki jih določajo metode. Orodja CASE imajo vgrajene mehanizme za preverjanje skladnosti in completeness običajno na tri načine. To so:

- aktivno, kar pomeni, da imajo vgrajen mehanizem za preprečevanje nedovoljenih relacij med simboli polformalnih modelov, ki nasploh onemogoča kreiranje takih relacij;
- pasivno, z vgrajenimi mehanizmi za identifikacijo neskladij, kar pomeni, da sicer ne

preprečujejo tistih nedovoljenih relacij med simboli, ki so že predefinirane, ampak le opozarjajo nanje (opozarjanje je lahko sprotno, to je med samo izdelavo modela ali pa, ko se zahteva preizkus skladnosti modela);

- pasivno, z uporabniško generiranimi mehanizmi za identifikacijo neskladij, kar je podobno kot v prejšnjem primeru, z edino razliko, da nedovoljene relacije definira uporabnik.



Sl. 8.16. Struktura sodobnih orodij CASE in njihova povezava z okoljem

Če imajo orodja CASE na razpolago vse tri možnosti, so zelo odprta in dovoljujejo uporabniku kreiranje lastnih relacijskih pravil. To pomeni, da lahko uporabnik definira metodo s svojimi pravili, s prilagoditvijo orodij pa si zagotovi preizkus skladnosti.

▽

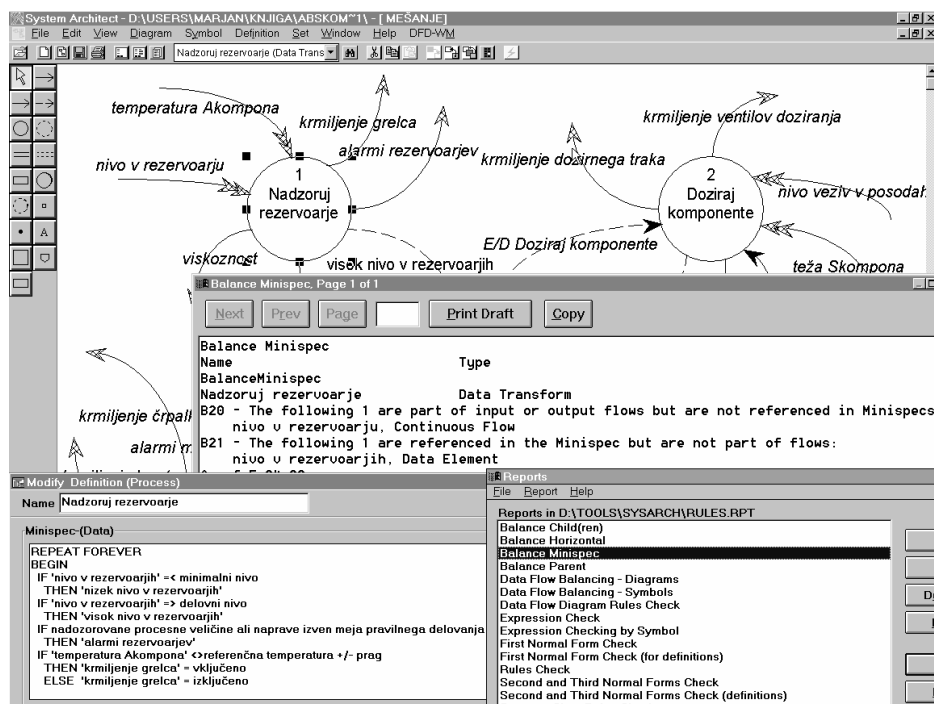
Primer 8.3: Uporaba orodja System Architect pri izdelavi specifikacij

Na naslednjih dveh slikah bomo ilustrirali del možnosti, ki jih nudijo sodobna orodja CASE v večokenskem okolju. S standardnim načinom dela, značilnim za okolje oken, je možno tako skoraj istočasno uporabljati bogat nabor polformalnih modelov - diagramov, preverjati njihovo skladnost in kompletnost, pregledovati ali kreirati podatkovne elemente in strukture ter generirati predefinirana ali posebna poročila o modelu in

procesu njegove izgradnje.

Predstavili bomo dva načina uporabe orodij: razvojno uporabo, to je uporabo orodij med kreiranjem modelov, ko se ti še spreminjajo; in pregled že izdelanih in preverjenih modelov.

Uporabo bomo prikazali na zgledu, ki je bil predstavljen v podpoglavju o programskem inženirstvu (7.5.1).



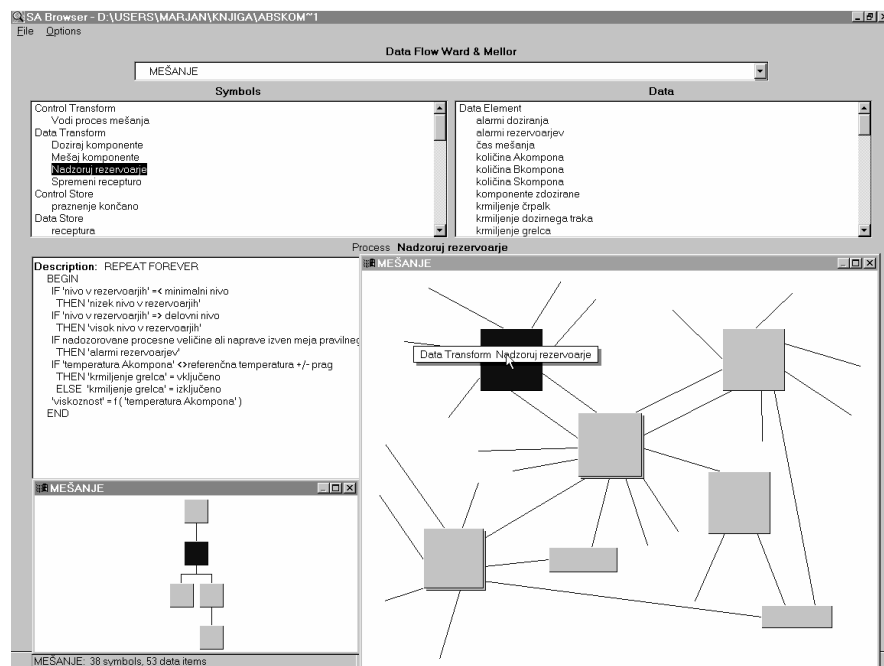
Sl. 8.17. Primer uporabniškega vmesnika sodobnega orodja CASE pri preverjanju notranje skladnosti diagrama tokov podatkov

Sl. 8.17 prikazuje primer izgleda zaslona pri uporabi sodobnega orodja CASE (System Architect) in sicer med razvojem modela pri preverjanju notranje skladnosti diagrama tokov podatkov.

Osnovna slika predstavlja polformalni model, ki je diagram tokov podatkov s sintakso in semantiko po metodi Ward-Mellor. Konkretno je to sistemski diagram za analizo enega od tehnoloških podprocesov. Predpostavimo, da je bil diagram pravkar narisani in opremljen z vsemi opisi podatkovnih in kontrolnih tokov ter podatkovnih in kontrolnih procesov. Naloga, ki jo bomo opravili s pomočjo orodja CASE, je preverjanje notranje skladnosti diagrama. Izbrali smo preverjanje skladnosti vstopajočih in izstopajočih tokov v proces *Nadzoruj rezervoarje* s specifikacijo procesa. Najprej smo izbrali proces, ki je zato na osnovnem oknu označen z majhnimi črnimi kvadrati. Nato smo izmed

poročil, katerih možen nabor je prikazan na dialogu "Reports", izbrali preverjanje skladnosti specifikacij ("Balance Minispec"). Po aktiviranju preverjanja se je na zaslonu odprlo okno "Balance Minispec", kjer je izpisano poročilo preverjanja. V poročilu so izpisane vse ugotovljene neskladnosti. V našem primeru opazimo, da obstajajo neskladnosti med imenom vhodnega toka podatkov *nivo v rezervoarju* in uporabo tega toka v pisni specifikaciji (*nivo v rezervoarjih*), ki je prikazana v oknu "Modify Definition". Po odpravi neskladnosti bo ponovna aktivacija preverjanja dala rezultat brez napak.

Primer pregledovanja skladnega modela z orodjem CASE je ilustriran s Sl. 8.18.



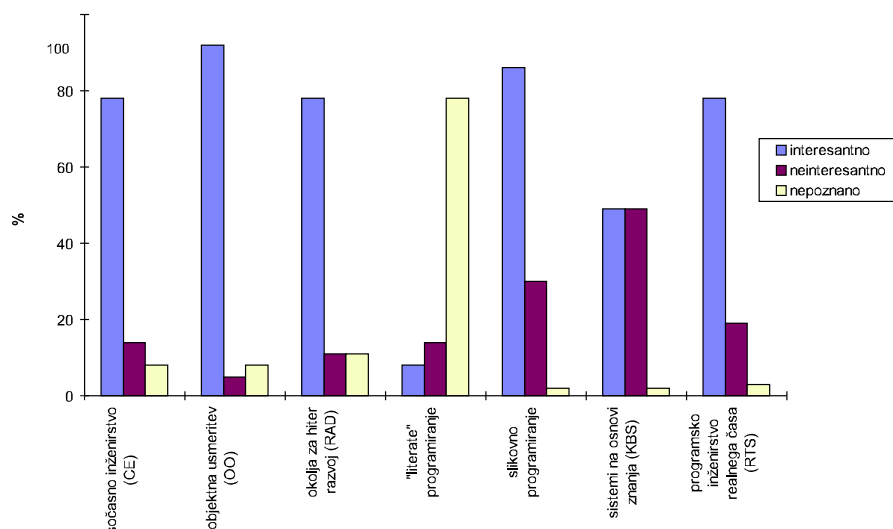
Sl. 8.18. Primer uporabniškega vmesnika sodobnega orodja CASE na zaslonu pri pregledovanju modelov

Ta način je uporaben, kadar želimo le pregledovati modele, ne pa jih spreminjati. Ponovno je uporabljen zgled iz poglavja o programskem inženirstvu. Vzemimo, da želimo pregled določenih podatkov, za katere pa ne vemo natančnih imen, niti v kateremu diagramu nastopajo. Za hiter dostop do takih podatkov je zelo primeren dostop preko drevesne strukture diagramov. Na sliki je ta prikazana v spodnjem levem oknu. V strukturi smo izbrali drugi, to je sistemski nivo oziroma sistemski diagram tokov podatkov *MEŠANJE*. Takoj po izboru se je odprlo novo okno (spodnje desno), v katerem so shematsko prikazani vsi elementi diagrama. Hkrati pa se v osnovnem oknu prikažejo vsi simboli in podatki, ki nastopajo na diagramu. Po podatkih lahko poizvedujemo v osnovnem oknu z izborom imena določenega podatka ali v z izborom

simbola na shematskem diagramu. V našem primeru smo izbrali iz zbirke simbolov v gornjem levem okvirju osnovnega okna podatkovni proces *Nadzoruj rezervoarje*. Rezultat poizvedbe je specifikacija tega procesa, prikazana v spodnjem okvirju osnovnega okna.

Δ

Predvideva se, da bodo na področje CASE vse bolj vplivali sodobni koncepti razvoja sistemov in novi načini implementacije programske opreme. Za ilustracijo tega je primerna Sl. 8.19, ki prikazuje rezultate raziskave mnenj uporabnikov o bodočih usmeritvah programskega inženirstva (Holt, 1997).



Sl. 8.19. Koncepti, ki bodo imeli vpliv na bodoče usmeritve CASE

Uvajanje orodij CASE

Orodja CASE je smiselno uvesti, ko se doseže definirana stopnja razvoja programske opreme. To je, ko se v procesu razvoja spoštuje vsaj interne standarde, upošteva organizirano vodenje sprememb in, ko je v vse aktivnosti vključeno zagotavljanje kakovosti. Proizvajalci orodij CASE razglašajo raznovrstne prednosti, ki naj bi jih orodja nudila. Tako so bodoči uporabniki večkrat zmedeni, ko ugotovijo neskladnosti med trditvami proizvajalcev in izjavami dejanskih uporabnikov. To je mnogokrat vzrok, da se ne odločijo za delo z orodji CASE, večkrat pa imajo zato celo nezaupanje v metode in metodologije. Nasprotovanja uvajanju temeljijo predvsem na subjektivnih razlogih. Orodja CASE zahtevajo predpisan način dela. Vsak posameznik je naravnan na lasten način dela, zato zaradi drugačne miselne naravnosti lahko pride v začetku do

zmanjšanja učinkovitosti dela. Obstaja tudi miselnost, da orodja CASE zaradi predpisanih postopkov, zmanjšujejo ustvarjalnost. Stališče izkušenih uporabnikov orodij CASE je, da naj pri manj obsežnih projektih vsak posameznik izbira njemu lastne načine dela. Nasprotno pa se pri obsežnih projektih izkaže, da sta metodološka podpora in uporaba orodij CASE nujna. S tem ustvarjalnost ni ogrožena, saj orodja le pomagajo pri opraviilih, ki sploh niso ustvarjalna, zahtevajo pa veliko časa. Čas lahko veliko bolj koristno izrabimo, če se osredotočimo na probleme, katerih reševanje dejansko terjaja ustvarjalni pristop.

Izbira ustreznega orodja CASE navadno zahteva kompromis med upoštevanjem različnih meril, kot so:

- CASE orodje izberemo glede na metodologijo, katero pri svojem delu že uporabljamo. Če je še ne uporabljamo, moramo najprej izbrati metodologijo, ki se bo najbolj prilegala reševanju naših problemov, jo osvojiti, in šele potem poiskati najprimernejše orodje CASE, ki to metodologijo podpira;
- izberemo takšno orodje, za katerega imamo ustrezno računalniško opremo, tako glede na vrsto, kot tudi glede na procesno moč;
- izberemo proizvajalca. Pri tem moramo vedeti ali proizvajalec dobavlja le orodja, ki jih potrebujemo, in ali omogoča nadalje širjenje orodij. Pomembno je tudi, da proizvajalec nudi izobraževanje, ter da lahko z njim vzpostavimo stik ob možnih težavah pri delu z orodji;
- v primeru, da izberemo orodje, ki ima možnost avtomatskega generiranja kode, moramo vedeti za kateri programski jezik naj orodje generira kodo in v kakšnem obsegu;
- če se načini dela ali pa metodologije, ki jih uporabljamo, razlikujejo od tistih, ki jih orodje podpira, moramo izbrati orodje, ki je odprto za prilagoditve;
- če pri delu že uporabljamo orodja CASE, radi pa bi njihovo uporabo razširili, moramo izbrati orodje, ki bo skladno s tistim, ki ga že imamo;
- izberemo takšno orodje, ki nam v največji meri omogoča predstavitev značilnosti sistemov, ki so tipični v naših projektih.

8.3.1.2 Uporaba orodij CASE v projektih za vodenje procesov

Zahteve za orodja CASE za vodenje procesov

Programska oprema za vodenje procesov spada v programsko opremo, ki mora izpolnjevati zahteve po izvrševanju svojih funkcij v realnem času. Zahteve za tovrstno programsko opremo in njene značilnosti so opisane v devetem poglavju, zato naj na tem mestu poudarimo le, da se te zahteve preslikajo tudi v zahteve razvojnih orodij, med katere spadajo tudi orodja CASE. Bistvena razlika med orodji CASE, ki podpirajo življenjski cikel poslovno-informacijskih sistemov in orodji, ki podpirajo sisteme za vodenje procesov je, da morajo v slednjem primeru orodja zagotavljati podporo ne le podatkovnemu in procesnemu vidiku sistemov, ampak tudi dinamičnemu, to je vidiku

časovnega obnašanja sistemov. Dalje to pomeni, da morajo ta orodja podpirati metode in metodologije, v katere so naštetih mehanizmi že vgrajeni (Rodd, 1995). Dolga leta so bile zato najprimernejše strukturne metode z razširitvijo za sisteme realnega časa, v zadnjem času pa jih vse bolj izpodrivajo objektno usmerjene metode.

Za uporabo v projektih vodenja procesov je primernih večina današnjih orodij CASE, ker podpirajo modeliranje obnašanja sistemov realnega časa ter nudijo pomoč pri načrtovanju, izvedbi in preizkušanju zahtevane programske opreme.

Orodja CASE, ki so primerna za izdelavo funkcionalnih specifikacij programske opreme procesnih sistemov, uporabljajo modele, ki omogočajo predstavitev časovnega izvajanja posameznih opravil. V splošnem so specifikacije sistemov realnega časa sestavljene iz modelov, kateri so dobljeni na podlagi usklajevanja zahtev in potreb uporabnika z izvajalcem in iz standardnih predefiniranih funkcij, ki so lastne posameznim vrstam sistemov realnega časa. V sistemih za vodenje procesov so predefinirane funkcije na primer algoritmi vodenja ipd. Specifikacije podatkovnih struktur in opravil, ki so značilni za specifično področje uporabe, lahko tudi shranimo in jih kasneje ponovno uporabimo na drugih projektih, kot smo to podrobneje pojasnili pri razlagi domenskega inženirstva.

V fazi načrtovanja programske opreme nekatera orodja CASE omogočajo združitev modulov programske opreme, ki so zgrajeni na osnovi uporabnikovih specifikacij, z operacijskimi sistemi za delo v realnem času. Simulacija izvajanja modela kompletne programske opreme je možna s posebnimi orodji, imenovanimi simulatorji dinamičnega obnašanja. Z njimi je možno preveriti, če načrtovana programska oprema zadošča časovnim zahtevam.

Naslednja orodja za izdelavo programske opreme za sisteme realnega časa so tako imenovana prototipna orodja. Namenjena so predvsem za graditev programske opreme za komuniciranje uporabnika z računalnikom preko zaslona. V projektih procesnega vodenja so zelo uporabna, saj je ravno komunikacija med uporabnikom in procesom bistvenega pomena za uspešno vodenje. Vsebujejo vnaprej določene oblike zaslonskih sporočil, ki pa jih lahko spreminjamo po želji uporabnika. Orodja omogočajo, da se programska oprema, ki podpira izbrano obliko sporočil, v CASE okolju avtomatsko vključuje v ostalo namensko programsko opremo. Nekaj več o njih bomo povedali v naslednjem podpoglavju.

Orodja za prototipno generacijo kode, namenjene za sisteme realnega časa, generirajo kodo v programskih jezikih kot so npr. Pascal, ADA in C. Avtomatsko generirano kodo s pomočjo simulacijskih kontrolnih jezikov, preiskujemo na simulatorjih realnih objektov. Na tak način so lahko generirani le neodvisni in enostavni programski deli, medtem ko v splošnem avtomatska generacija celotne izvršljive kode za sisteme realnega časa direktno iz specifikacij, danes še ni možna.

Zahteve zaradi načina izdelave projektov za vodenje procesov

Projekti izgradnje sistemov za vodenje procesov oziroma najširše rečeno projekti računalniške avtomatizacije in informatizacije se mnogokrat izvajajo paralelno z drugimi projekti, npr. strojnimi, gradbenimi, itd. Zato ne sestojijo le iz računalniškega podprojekta, v katerem so zajeti koncepti vodenja, programska in aparturna oprema, ampak tudi iz drugih podprojektov. Ker izvajalci posameznih podprojektov pri svojem delu uporabljajo različna računalniška orodja, morajo biti vsa orodja med seboj združljiva. Le na ta način je omogočena skladna uporaba istih podatkov v različnih podprojekti in sledljivost izvajanja projektnih aktivnosti. Orodja CASE morajo biti zato neposredno povezljiva z ostalimi orodji ali pa preko sistema "import/export", preko enake podatkovne baze ali na kak drug skladen način.

Ker je v razvojnih fazah projektov računalniške avtomatizacije in informatizacije praviloma udeleženo veliko število raznovrstnih strokovnjakov, tako s strani naročnika kot s strani izvajalca, je potrebno metode, dokumentacijo in uporabniške vmesnike orodij CASE prilagoditi tako, da so hitro in enoumno razumljivi za celotno projektno ekipo.

Iz zgoraj navedenega izhaja, da morajo orodja CASE omogočati prilagajanje na konkretne zahteve uporabnika, kar pa je možno le, če so dovolj odprta in dovolj podprta z ustrežno dokumentacijo.

8.3.2 Orodja za hiter razvoj aplikacij

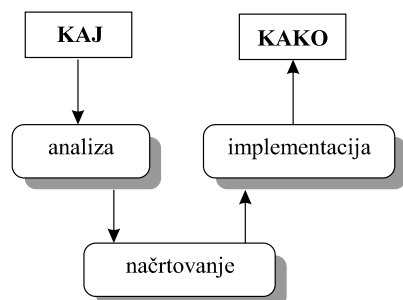
Osnovni smisel CASE orodij, o katerih smo govorili v prejšnjem delu poglavja je seveda povečati učinkovitost in kvaliteto razvoja programske opreme. V tem delu poglavja pa bomo predstavili posebno skupino sodobnih orodij, ki omogočajo še dodatno skrajševanje razvoja. To so orodja, ki podpirajo prototipni pristop k razvoju programske opreme.

8.3.2.1 Fazni in prototipni pristop

Ob nastanku prvih računalnikov so programerji, posredno pa tudi uporabniki opazili, da obstaja semantična in sintaktična vrzel med navodili za delo, ki bi jih želel posredovati človek in navodili za delo, ki bi jih razumel računalnik. V splošnem človek teži h temu, da definira KAJ naj bi bilo storjeno, stroj pa pričakuje navodila o tem KAKO naj bi bilo določeno delo opravljeno. Naslednji problem pa je povezan z dejstvom, da naročnik ob izstavitvi naročila običajno ne ve točno kaj želi, tega ne zna enolično definirati, ali pa, kar je še bolj pogosto, da se njegove zahteve spreminjajo s časom zaradi zunanjih vzrokov. Slednje vodi k izdelavi niza zelo podobnih proizvodov, ki se s stališča naročnika le minimalno razlikujejo, na programerski (in računalniški strani) pa predstavljajo bistveno spremembo izdelane kode.

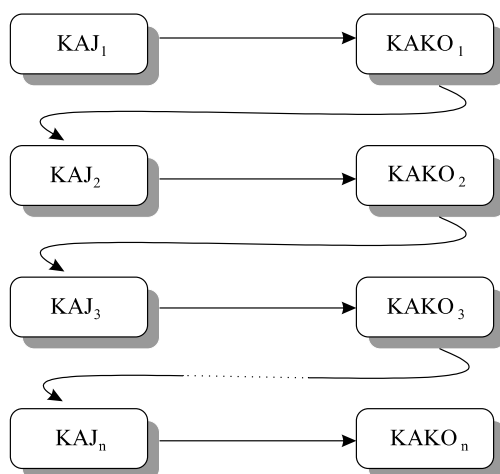
Trenutno najbolj uveljavljen pristop poskuša kvalitativni prehod med KAKO in KAJ premostiti s fazami življenjskega cikla (glej Sl. 8.20). Velik razkorak se razdeli na niz

zaporednih faz, pri čemer je semantična vrzel med posameznimi fazami manjša. Fazni pristop ima dve bistvene pomanjkljivosti: relativno dolg čas razvoja prve verzije produkta ter nezmožnost brezšivne integracije faz življenjskega cikla.



Sl. 8.20. Fazni pristop

Alternativa faznemu je prototipni pristop (Martin, 1991) (glej Sl. 8.21). Prototipni pristop predpostavlja, hitro izvedbo prvega prototipa ter kasnejše izboljševanje le tega v interakciji z uporabnikom. Prototipnega pristopa si ne moremo zamisliti brez interaktivnega dela z računalnikom in s pomočjo zmogljivih orodij, ki grafično prikazujejo relacije med komponentami izdelka, generirajo programsko kodo, omogočajo izdelavo uporabniškega vmesnika, itd. Prototipni pristop je zaživel s prihodom osebnih računalnikov, ki omogočajo delo na relativno ceneni, a zmogljivi platformi z grafičnim vmesnikom ter zmogljivim procesorjem in diskom.



Sl. 8.21. Prototipni pristop

V osnovi ločimo dva pristopa pri izdelavi prototipov (Gordon, Bieman, 1994):

- *evolucijski prototipi* (Evolutionary prototypes) omogočajo koračno izboljševanje ciljne programske opreme, ki na koncu običajno rezultira v končnem produktu, t.j. "zadnji prototip" je sočano tudi izdelek, ki se preda naročniku;
- *prototipi za enkratno uporabo* (Throwaway prototypes) omogočajo bolj natančno definicijo zahtev naročnika in verifikacijo pomembnejših rešitev v fazi oblikovanja. Prototip je namenjen zbiranju informacij in se na koncu zavrže.

V zadnjem času se je za tovrsten pristop uveljavilo ime RAD (Rapid Application Development), oziroma okolja za hiter razvoj aplikacij. Definicij je več. Poglejmo si dve.

Prva definicija se glasi (Internet, 1998):

RAD je metodologija, ki omogoča razvoj strateško pomembnih aplikacij hitreje, pri čemer zmanjšuje razvojne in vzdrževalne stroške ob sočasnem enakem nivoju kvalitete. Cilje je možno doseči z nizom razvojnih tehnik v okviru dobro definirane metodologije. Tehnike vključujejo uporabo:

- *majhnih, dobro izšolanih timov strokovnjakov;*
- *evolucijske prototipe;*
- *integrirana orodja za podporo modeliranja, prototipiranja in ponovne uporabe komponent;*
- *centralno skladišče podatkov (o strukturi aplikacije);*
- *okolja za interaktivno definicijo zahtev in oblikovanje rešitev;*
- *natančne časovne omejitve izdelave programskih podsklopov.*

Druga definicija pa pravi (Kinmond, 1995):

RAD ni metodologija ampak oblika uporabe faznega pristopa, ki rezultira v krajšem razvojnem ciklu ter višji kvaliteti končnega produkta. Poudarek je na kvaliteti kot jo definira naročnik, v smislu izdelave pravega produkta že pri prvi verziji in produkta, ki ustreza resničnim potrebam naročnika.

V nadaljevanju se bomo omejili na opis pričakovanih lastnosti RAD orodja za razvoj aplikacij na procesnem nivoju.

8.3.2.2 Pomembne lastnosti RAD orodij

RAD orodja, oziroma programi, ki jih s pomočjo RAD orodij zgradimo, imajo niz lastnosti, ki so bolj ali manj pomembne za razvoj aplikacij na procesnem nivoju. Med njimi so najpomembnejše:

Prijaznost in hitrost izdelave uporabniškega vmesnika

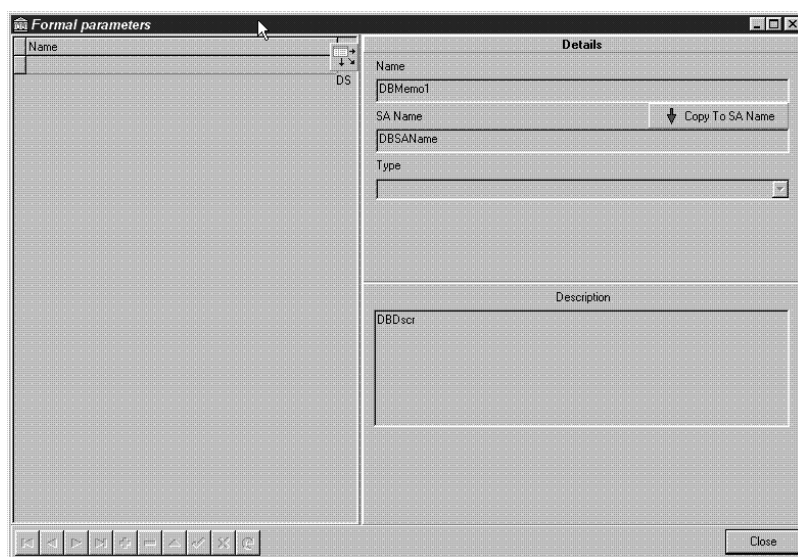
Pri uporabi klasičnih orodij je izdelava prijaznih uporabniških vmesnikov dolgotrajen in

mukotrpen posel. Množica obravnavanih dogodkov izredno poveča obseg programske kode, ki lahko postane nepregledna. Če je izdelava uporabniškega vmesnika podprta z avtomatskim generiranjem programske kode ali še bolje s knjižnico razredov, se razvojni čas zmanjša, manj je napak, koda pa postane bolj pregledna.

▽

Primer 8.4: *Izdelava uporabniškega vmesnika*

Sl. 8.22 je primer izdelave kompleksnega uporabniškega vmesnika, ki vključuje tabelarični prikaz, eno in večvrstična vnosna polja ter gumbe. Prikazani primer ni zahteval niti ene vrstice ročno kodirane kode. S klasičnim pristopom, bi ekvivalentna funkcionalnost zatevala vsa 500 vrstic izvirne kode.



Sl. 8.22. *Izdelava uporabniškega vmesnika*

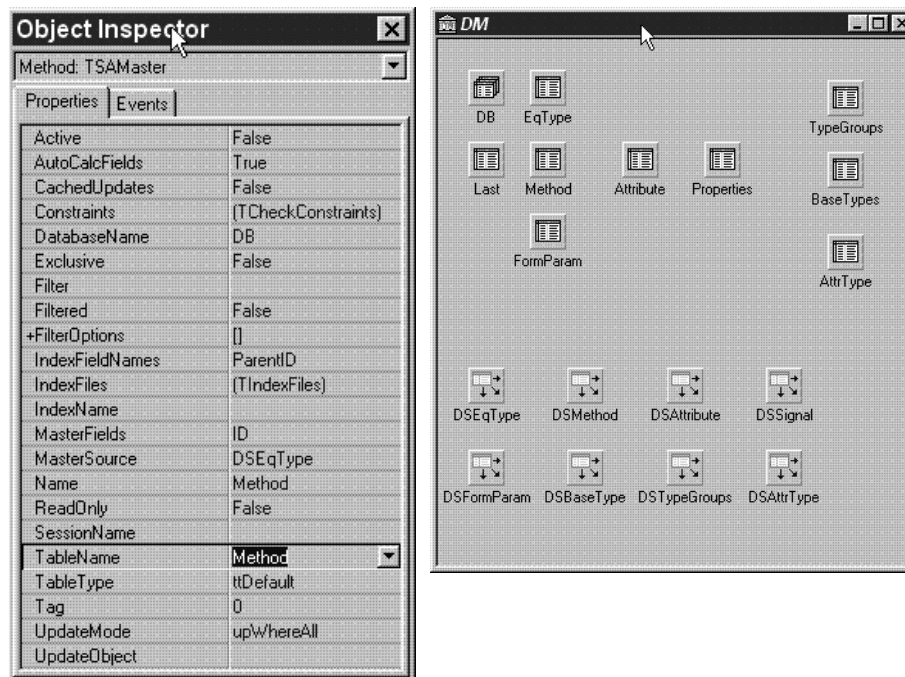
△

Vmesniki do baz podatkov (predvsem ODBC)

Procesne baze podatkov so pomemben element kompleksnejših sistemov za vodenje. Programski dostop do njih mora biti učinkovit in enostaven. ODBC je splošno sprejet standard za tovrstno dostopanje. Če RAD orodje podpira razrede, ki združi klice ODBC, API in rutinska opravila (vzpostavitev povezave, seje, transakcij, itd) nad podatkovno bazo, je implementacija lažja in hitrejša. Smiselno je tudi povezovanje z orodji za grafično načrtovanje baz podatkov, ponovno s ciljem minimiziranja števila “ročno sprogramiranih” vrstic kode.

▽

Primer 8.5: Definiranje baze podatkov



Sl. 8.23. Definiranje baze podatkov

Na Sl. 8.23 vidimo primer definicije baze podatkov, do katere ima aplikativni program dostop. Desna slika prikazuje bazo podatkov sestavljeno iz devetih tabel ter istega števila pogledov (view) nanje. Leva slika pa prikazuje seznam lastnosti, ki jih lahko definiramo pri vsaki tabeli.

△

Podpora “OLE Automation” tehnologiji

Podobno vlogo kot jo igra ODBC na področju baz podatkov, predstavljajo tehnologija “OLE Automation” in njeni predhodniki pri izdelavi vmesnikov za komunikacijo z drugimi podsčkopi, oziroma celih podsčkopov uporabniškega vmesnika. Tipični primeri uporabe so npr. okna za prikaz grafov, dostop do Interneta (http, mail), posebni numerični vnosni formati, prikaz vsebine podatkovne baze na zaslon, itd. OLE Automation (podobno kot konkurenčni standard CORBA) predstavlja osnovo nove evolucijske stopnje pri razvoju programske opreme. Komponente (orig. “component ware”) omogočajo izgradnjo sistemov na podlagi črnih škatel z znanim obnašanjem.

Zato je seveda logično pričakovati, da RAD orodja podpirajo tako izdelovanje lastnih komponent kot tudi možnost uporabe tujih komponent.

Kvaliteta navodil

Ker je potencialni uporabnik orodja RAD tudi človek, ki ni strokovnjak na računalniškem področju, je pomembno, da obstajata dva tipa navodil:

- uporabniški priročnik (user's guide);
- pregledni priročnik (reference manual).

Prvi poskrbi za prikaz osnovnih konceptov in mora korakoma podati nekaj konkretnih primerov uporabe. Drugi poskrbi za programerja, saj mora vsebovati natančen opis "API-ja" ter fragmentiran prikaz uporabe.

Stabilnost produkta

Večina programske opreme prihaja na tržišče še preden je dodobra očiščena vseh napak. V nekaterih primerih je teh napak, več v nekaterih manj, nekatere so bolj nekatere pa manj moteče. Uporabnik produkta seveda teži h temu, da je napak čim manj, in da so čim manj moteče.

Regeneriranje obstoječe kode

Ena od značilnosti današnjih RAD orodij je relativno velika količina avtomatsko generirane kode, bodisi na podlagi manipulacij z objekti, ali pa s pomočjo t.i. "čarovnikov" (t.i.. Wizards). Pomembno je, če zna orodje samo detektirati kodo, ki jo je samo zgeneriralo in jo po potrebi ustrezno modificirati.

Orodja RAD kot evolucijski korak v procesu razvoja okolij za razvoj programske opreme so že pokazala svoje prednosti in slabosti in opravičujejo svoj obstoj tudi v za njih v začetku tujih okoljih. Zato lahko pričakujemo njihov nadaljnji razvoj in uveljavljanje tudi na področju sistemov za vodenje procesov.

8.3.3 Programski jeziki za sisteme realnega časa

Sistemi realnega časa morajo istočasno izvajati različna asinhrona in sinhrona opravila, od vzorčenja, ugotavljanja trenutnega stanja, do izvajanja ukrepov vodenja. Pri teh sistemih se skoraj vedno pojavljajo višje zahteve glede zanesljivosti in predvsem varnosti, kot je to primer pri drugih računalniških sistemih. Pri doseganju tega cilja so nedvomno ključnega pomena pravilne specifikacije, ustrezna metoda načrtovanja ter dobra razvojna orodja. Seveda pa pri izpolnjevanju teh zahtev ni vseeno, kakšen programski jezik uporabimo. Če naj jezik podpira učinkovito načrtovanje, razvoj in vzdrževanje programske opreme za delo v realnem času, mora poleg vseh lastnosti, ki jih morajo imeti splošni programski jeziki, zadoščati nekaterim zahtevam, ki izvirajo iz

posebnosti sistemov realnega časa.

8.3.3.1 Zahteve jezikov za sisteme realnega časa

Glavne zahteve, ki naj bi jih izpolnjevali programski jeziki za sisteme realnega časa so zanesljivost, varnost, ponovna uporabljivost, možnost vzdrževanja, čitljivost, fleksibilnost, učinkovitost, enostavnost in prenosljivost. V nadaljevanju tega poglavja bomo definirali te zahteve ter našli lastnosti jezikov, ki najbolj prispevajo k zadovoljevanju posameznih zahtev. Važnejše lastnosti jezikov bomo podrobneje obravnavali v nadaljevanju.

Zanesljivost

Zanesljivost (*security*, včasih tudi *reliability*) programskega jezika moramo razlikovati od podobnih pojmov, ki se nanašajo na programsko opremo v splošnem, kot so korektnost (*correctness*) in zanesljivost (*reliability*). Korektnost je statična lastnost programske opreme, ki izraža njeno konsistentnost s specifikacijami, torej je to lastnost, ki jo ugotavljamo z verifikacijo. Zanesljivost je dinamična lastnost programske opreme, ki izraža stopnjo verjetnosti, da bo program zadovoljivo izvajal koristne akcije takrat, ko se to od njega pričakuje.

Po drugi strani pa zanesljivost programskega jezika definiramo kot možnost preprečevanja in odkrivanja napak. Preprečevanje napak je najbolj odvisno od izrazne moči, ki jo ponuja jezik pri preslikavi problemske domene v programsko domeno. Ta izrazna moč izvira predvsem iz mehanizmov abstrakcije, ki jih jezik vsebuje. Osnovni mehanizmi abstrakcije so podatkovni (podatkovni tipi) in funkcionalni (procedure in funkcije). Bolj kompleksni mehanizmi abstrakcije so objektni, ki združujejo (*enkapsulacija*) podatke in operacije, ki se nad njimi izvajajo (*abstraktni podatkovni tipi*), v entitete, ki okolju dovoljujejo dostop do podatkov le na omejen in vnaprej določen način (*skrivanje informacij*). Zelo važen mehanizem abstrakcije je tudi procesna abstrakcija, ki daje veliko izrazno moč pri preslikavi inherentne sočasnosti sistemov realnega časa v programsko domeno.

K možnosti odkrivanja napak (pri prevajanju) največ prispeva način obravnavanja podatkov v smislu omejevanja obsega vrednosti in dovoljenih operacij (strogo obravnavanje tipov, omejevanje vidljivosti podatkov), kontrolne strukture programskega toka (strukturiranost jezika) ter modularnost jezika in način prevajanja programa.

Ostali faktorji, ki vplivajo na zanesljivost jezika, so: dobro definiran standard, dobro definirano obnašanje in preverjanje porabe pomnilnika

Varnost

Varnost (*safety*) programskega jezika je direktno povezana s pojmom varnosti programske opreme, ki jo definiramo kot stopnjo verjetnosti, da programska oprema ne bo povzročila kaj nevarnega. Varnost torej obravnava posledice dejstva, da je

programska oprema nezanesljiva. Na varnost programskega jezika najbolj vpliva možnost sistematičnega odkrivanja in obravnavanja napak pri izvajanju programa (*exception handling*). Pri kritičnih sistemih, kjer so ekstremno visoke zahteve glede varnosti, je zaželen obstoj podmnožice jezika za kritične aplikacije, pri kateri so izločeni vsi elementi jezika, ki zmanjšujejo njegovo zanesljivost.

Ponovna uporabljivost

Ponovna uporabljivost (reusability) je, kot smo že omenili v prejšnjem poglavju, uporaba gradnikov, postopkov ali znanja iz obstoječih sistemov pri gradnji novih. Analiza, načrtovanje in implementacija programske opreme za ponovno uporabljivost (design-for-reuse) omogočajo bistveno povišanje kvalitete in znižanje cene izdelave programske opreme. Med lastnosti jezikov, ki največ prispevajo k ponovni uporabljivosti, lahko uvrstimo razne vrste abstrakcij in modularnost.

Možnost vzdrževanja

Praktično vsako programsko opremo, ki se uporablja, je potrebno spreminjati. Spremembe so potrebne zaradi sprememb v zahtevah ali zaradi popravljanja napak. Pri vzdrževanju praviloma prihaja do povečanja entropije (nereda) programske opreme. Možnost vzdrževanja (maintainability, modifiability) je obratno sorazmerna s tem povečanjem entropije. Če predpostavimo, da je povečanje entropije ob spreminjanju proporcionalno z entropijo pred spreminjanjem, je v bistvu najboljši način zagotavljanja možnosti vzdrževanja, graditev sistema s čim nižjo začetno entropijo. K temu najbolj prispevajo strogo obravnavanje tipov, omejevanje vidljivosti podatkov, strukturiranost jezika, modularnost jezika in razne vrste abstrakcij.

Čitljivost

Ni treba posebej poudarjati važnosti dobre čitljivosti programa. Le-ta omogoča zniževanje stroškov dokumentiranja in vzdrževanja ter lažje odkrivanje napak, kar povečuje zanesljivost. Lastnosti jezika, ki vplivajo na čitljivost, so strukturiranost jezika, njegova modularnost ter nenazadnje njegova sintaksa, ki določa npr. izbiro ključnih besed jezika, omejitve pri imenih identifikatorjev, jasnost matematičnih simbolov in pravila glede oblike izvorne kode.

Fleksibilnost

Programski jezik je fleksibilen, če omogoča izvajanje vseh potrebnih (tudi nizkonivojskih) operacij brez uporabe zbirnega jezika (ki je inherentno nezanesljiv), po drugi strani pa mora tudi omogočati vključevanje segmentov kode v zbirnem jeziku (npr. pri časovno kritičnih odsekih). Fleksibilnost je izjemno važna lastnost, saj moramo pri sistemih realnega časa pogosto obravnavati prekinitve, dostopati do različnih perifernih enot, do absolutnih naslovov v pomnilniku ali do posameznih bitov.

Učinkovitost

Učinkovitost v klasičnem pomenu besede izraža značilnost jezika, da je njegova objektna koda kompaktna in hitra za izvajanje. Jezik mora zato vsebovati take konstrukte, ki omogočajo učinkovito implementacijo. Zaradi zagotavljanja odzivnih časov sistema se je potrebno izogibati mehanizmom, katerih čas izvajanja je nepredvidljiv. V preteklosti je bila učinkovitost jezika obravnavana kot najvažnejša, kar je šlo na račun zanesljivosti, fleksibilnosti in enostavnosti jezika.

Determinističnost

Danes, ko je materialna oprema vedno zmogljivejša, mi pa se vedno bolj zavedamo pomembnosti časovno determinističnega obnašanja sistema (ne čim hitreje, ampak pravočasno izvajanje funkcij), pomembnost učinkovitosti programskega jezika v klasičnem smislu pada. Pojavlja pa se potreba po možnosti generiranja časovno deterministične kode z možnostjo vnaprejšnjega ugotavljanja časa izvajanja, nastavljanja raznih mejnih časov in izbire strategije razporejanja dejavnosti pri večopravilnih aplikacijah. Čeprav so te zahteve še posebej važne pri časovno kritičnih aplikacijah, sploh pa pri tistih, kjer nedeterministično obnašanje programa lahko izzove materialno škodo ali celo izgubo človeških življenj, jih danes še noben programski jezik ne podpira.

Enostavnost

Enostavnost jezika omogoča lažje učenje in uporabo, kar dosežemo z izogibanjem kompleksnim konstruktom, še bolj pa z izogibanjem raznim nelogičnim omejitvam v jeziku (konsistentnost pravil, uniformnost).

Prenosljivost

Prenosljivost pomeni neodvisnost programske od materialne opreme, kar omogoča prenos programov z enega na drug računalnik. V sistemih realnega časa je prenosljivost težko doseči, je pa tudi manj pomembna, kot pri nekaterih drugih področjih uporabe računalnikov. Na prenosljivost jezika vpliva dobro definiran standard in obstoj standardnih knjižnic.

8.3.3.2 Zahtevane lastnosti jezikov

Za zanesljivost jezika je gotovo zelo pomemben način obravnavanja podatkovnih struktur. Zelo pomembno je tudi strukturirano programiranje, zato mora jezik vsebovati konstrukte, ki omogočajo, ali celo silijo k strukturiranemu programiranju.

Sistemi realnega časa so pogosto veliki in kompleksni, pri njihovem razvoju sodelujejo večje ekipe, pogosto so potrebne spremembe. Iz tega izhaja zahteva po modularnem programiranju, omejenem dostopu do podatkov (in drugih virov) in ločenem prevajanju.

Zaradi čim boljše preslikave iz problemske v programsko domeno mora jezik imeti možnost abstraktnih podatkovnih tipov.

Zaradi inherentne sočasnosti sistemov realnega časa, mora programski jezik vsebovati elemente multiprogramiranja oz. procesno abstrakcijo ter mehanizme medprocesne komunikacije in sinhronizacije.

Zaradi raznih nestandardnih vhodno-izhodnih enot, s katerimi mora sistem realnega časa komunicirati, kakor tudi zaradi potrebe po sistemskem programiranju, mora jezik omogočati nizkonivojsko programiranje.

Če naj bo sistem realnega časa zanesljiv in varen, mora programski jezik vsebovati tudi konstrukte, ki omogočajo odkrivanje in obravnavanje napak med izvajanjem programa (exception handling).

Druge lastnosti jezikov, ki jih tukaj ne bomo podrobneje obravnavali, so dobro definiran standard, dobro definirano obnašanje, preverjanje porabe pomnilnika, definirane matematične operacije, sintaksa in pravila glede oblike izvorne kode, obstoj podmnožice za kritične aplikacije ter razumljiva standardna knjižnica.

Podatkovne strukture

Visokonivojski programski jezik podaja *abstrakten model*, ki na problemsko orientiran način definira podatke in nad njimi dovoljene operacije. To pomeni, da vsak podatek pripada določenemu tipu, ki določa nabor vrednosti, ki jih podatek lahko zavzame, kakor tudi nabor operacij, ki so za ta tip dovoljene.

Vsako spremenljivko je potrebno *deklarirati*, ker to omogoča prevajalniku preverjanje konsistentnosti in graditev simbolne tabele ter zagotavlja informacijo o potrebnem pomnilniškem prostoru. Deklaracije so lahko *implicitne* (ob prvem omenjanju v programu) ali *eksplicitne* (v začetnem, deklaracijskem delu programa). Implicitna deklaracija slabo vpliva na zanesljivost jezika, naj omenimo samo možnost napačnega črkovanja, ko prevajalnik napačno črkovano spremenljivko smatra za novo.

Glede *vidljivosti* spremenljivke delimo na globalne in lokalne. Lokalizacijo spremenljivk omogočata dva konstrukta: modul in procedura. Lokalizacija spremenljivk omogoča uporabo principa skrivanja informacij - vsakemu delu programa je skrito tisto, česar nujno ne rabi. S tem se izognemo nehotenemu oziroma nekontroliranemu dostopu do spremenljivk. Zahvaljujoč lokalizaciji lahko imamo v različnih delih programa različne spremenljivke z istimi imeni, ne da bi prihajalo do konflikta. Lokalizacija spremenljivk pozitivno vpliva na zanesljivost in možnost vzdrževanja.

Glede na *življenjsko dobo* spremenljivke delimo na statične, ki obstajajo ves čas izvajanja programa, in na dinamične, ki obstajajo samo v določenih fazah izvajanja programa in omogočajo prihranek pomnilniškega prostora.

Vsak programski jezik pozna nekaj *osnovnih* tipov, kot so npr.:

- celo število (*integer*);
- naravno število (*cardinal, unsigned*);

- realno število (*real*);
- logična spremenljivka (*boolean*);
- črkovna spremenljivka (*char*).

Poleg teh naj bi jezik poznal še:

- naštevne tipe, pri katerih določimo zalogo vrednosti iz simboličnih imen;
- delne ali podintervalne tipe, ki imajo omejeno zalogo vrednosti osnovnega ali naštevnega tipa;
- izpeljane tipe, ki omogočajo ločevanje spremenljivk, ki so istega osnovnega tipa, logično pa so različne;
- proceduralne tipe, ki omogočajo obravnavanje procedur kot spremenljivk, in so uporabne pri spreminjanju imen standardnim proceduram in pri realizaciji procesne abstrakcije (implementaciji procesov).

Za doseganje višjega nivoja abtrakcije so bistveni sestavljeni tipi, kot so:

- polje (*array*);
- množica (*set*);
- zapis (*record*);
- dinamične podatkovne strukture in kazalci (*pointer*).

Po načinu podajanja razlikujemo šibko ali močno podane tipe. Jezik ima močno podane tipe, če velja:

- vsaka spremenljivka pripada določenemu tipu;
- vsak tip določa nabor vrednosti in operacij;
- pri vsaki prireditveni operaciji mora tip prirejene vrednosti biti enak tipu spremenljivke, ki se ji prireja vrednost;
- vsak operator nad podatkom mora pripadati naboru operatorjev prirejenih tipu tega podatka.

Pri močnem načinu podajanja tipov lahko zahtevamo strukturno enakost ali imensko enakost.

▽

Primer 8.6: Program z močno podanimi tipi

Na Sl. 8.24 vidimo primer dela programa, ki ilustrira način močnega podajanja tipov.

TYPE	Števec: INTEGER;
	Nivo: INTEGER;
VAR	i: Števec;
	L: Nivo;
...	
...	
i:=L;	(* pri zahtevani imenski enakosti bo prevajalnik tukaj javil napako *)

Sl. 8.24. Primer močno podanih tipov

Δ

Strukturirano programiranje

Vsak program je možno napisati ob uporabi samo treh struktur za kontrolo toka. To so *zaporedje*, *izbira* in *ponavljanje*. To je celo imperativ; edina izjema je v primeru redkih izjemnih situacij, pa tudi takrat mora biti tehnika reševanja natančno določena in kontrolirana.

Poznamo dve vrsti izbire:

- enojna (*IF-THEN-ELSE* stavek, pri čemer poznamo tudi gnezdene **IF** stavke);
- večkratna (*ELSE-IF* in *CASE* stavek).

Ponavljanj poznamo več vrst:

- ponavljanje s po-preverjanjem pogoja (*REPEAT-UNTIL* konstrukt);
- ponavljanje s pred-preverjanjem pogoja (*WHILE-DO* konstrukt);
- ponavljanje s preverjanjem znotraj zanke (*LOOP-EXIT* konstrukt);
- ponavljanje z vnaprej določenim številom ponavljanj (*FOR-TO* konstrukt).

GOTO stavek ponuja zelo široke možnosti zlorab, kar lahko pripelje do zelo nezanesljivih programov. Zato je njegovo uporabo potrebno omejiti le na izjemne primere ali pa jo popolnoma prepovedati.

Modularnost in način prevajanja

Program lahko napišemo *monolitno*, to je v enem bloku. To je najenostavnejši in v nekaterih izjemnih primerih tudi najprimernejši način. To so primeri, ko program razvija ena oseba, in ko je program zelo majhen in enostaven. Glavni razlogi za to so, da bi večji monolitni program postal nerazumljiv, pri vsaki spremembi bi morali prevajati cel program, pri iskanju vsake napake bi morali pregledovati cel program, zaradi napak bi prihajalo do nepredvidljivih stranskih učinkov.

Drug način je *modularni* pristop, pri katerem program razdelimo na več manjših podprogramov, ki jih lahko razdelimo na več oseb. Pri tem pristopu so spremembe navadno lokalizirane v enem modulu, isto velja tudi za napake. Veliko manjša je nevarnost stranskih učinkov. Še vedno pa moramo pri vsaki spremembi prevajati cel

program. Poleg tega ima ta pristop še eno veliko pomanjkljivost. Pri tem načinu je nujna določena količina globalnih podatkov, ki jih lahko spreminja vsak programer. To lahko pripelje do hudih napak v primeru nekontroliranega in nedokumentiranega dostopa do teh podatkov.

Tretji pristop je *neodvisno prevajanje*. Pri tem je število globalnih podatkov bistveno zmanjšano (ali jih sploh ni), pri spremembah pa je potrebno prevajati samo modul, v katerem je prišlo do spremembe, kar skrajša čas, potreben za spremembe in še dodatno zmanjša verjetnost stranskih efektov. Edina pomanjkljivost tega načina je v tem, da se nekatere napake (npr. klic procedure iz drugega modula z napačnim številom parametrov) odkrijejo šele med linkanjem.

To pomanjkljivost odpravi t.i. pristop *ločenega prevajaja*, pri katerem moramo prevajalniku podati potrebno informacijo o ostalih modulih, kar mu omogoči odkrivanje takšnih napak.

Abstraktni podatkovni tipi (objektna abstrakcija)

Programsko opremo vedno skušamo načrtovati s pomočjo postopka zaporednega razčlenjevanja. To velja tako za načrtovanje podatkovnih kot tudi programskih struktur. Pri podatkovnih strukturah nam to omogočajo sestavljeni tipi, pri programskih strukturah pa večnivojsko gnezdenje procedur.

Vzporednost postopnega razčlenjevanja podatkovnih in programskih struktur pripelje do združevanja podatkov in operacij, ki se nad njimi izvajajo, v programske enote, ki okolju dovoljujejo dostop do podatkov samo preko teh operacij. Tako programsko enoto imenujemo *modul*, princip zapiranja podatkov in operacij v modul imenujemo *enkapsulacija*, omejeni dostop do podatkov pa imenujemo *skrivanje informacij*. Modul je razdeljen na dva dela:

- *definijski del* določa kaj modul dela in kaj je dostopno okolju;
- *implementacijski del* določa kako modul dela in je od zunaj *neviden*.

Drugi moduli lahko uporabljajo samo tiste entitete (npr. proceduro, tip ali spremenljivko), ki so navedene v definijskem delu modula, in to šele takrat, ko to entiteto eksplicitno uvozijo (importirajo) v svojem deklaracijskem delu. Koncept modula ima dve zelo važni posledici:

1. omogoča graditev programskih knjižnic in prej omenjeno ločeno prevajanje. Pri tem je za prevajanje določenega modula dovolj napisati definijske dele modulov, ki jih ta modul uporablja;
2. omogoča definiranje *abstraktnih podatkovnih tipov* (ADT). ADT je tip, katerega ime je navedeno v definijskem delu modula (poleg operacij, ki so možne nad tem tipom). Struktura tega tipa je, enako kot implementacija operacij, skrita v implementacijskem delu. Na ta način imamo pravo abstrakcijo, saj npr. uporabnika, ki preko klica procedure *Create(VAR Q: Queue)* v modulu *PriorityQueue* kreira

vrsto procesov in potem npr. s klicem procedure *Enqueue*(VAR Q: Queue; P: ProcId) vstavi proces v vrsto “ready” procesov, sploh ne zanima, ali je vrsta implementirana npr. kot krožno polje, kopica ali lista (to se lahko celo med implementacijo modula spreminja, ne da bi vplivalo na module, ki uporabljajo abstrakcijo vrste). Poleg tega modul *PriorityQueue* lahko kreira poljubno število vrst na zahteve raznih uporabnikov, edina omejitev je razpoložljiva količina pomnilnika.

▽

Primer 8.7: Modul, ki opredeljuje prioritetno vrsto procesov

Primer modula, ki opredeljuje prioritetno vrsto procesov lahko vidimo na listingu na Sl. 8.25.

```
(*=====*)
DEFINITION MODULE PriorityQueue;
FROM TypeDefMod IMPORT ProcId;
EXPORT Queue, Create, Enqueue, Dequeue;
TYPE Queue;
PROCEDURE Create(VAR q:Queue);
PROCEDURE Enqueue(VAR q:Queue;p:ProcId);
PROCEDURE Dequeue(VAR q:Queue;VAR p:ProcId);
END PriorityQueue;
(*-----*)
IMPLEMENTATION MODULE PriorityQueue;
. . .
END PriorityQueue;
(*=====*)
IMPLEMENTATION MODULE Scheduler;
. . .
FROM PriorityQueue IMPORT Queue, Create, Enqueue, Dequeue;
VAR ReadyQueue:Queue;
. . .
PROCEDURE Dispatch;
. . .
    Enqueue(ReadyQueue, Processes[CurrentProcess].Id);
. . .
END Dispatch;
. . .
BEGIN
. . .
    Create(ReadyQueue);
. . .
END Scheduler;
(*=====*)
```

Sl. 8.25. Primer modula: Prioritetna vrsta procesov

Δ

Koncept ADT podpirajo takoimenovani objektno bazirani jeziki. Objektno orientirani jeziki temu dodajajo še koncepte razreda, objekta, kot instance (pojavitve) razreda,

dedovanja in polimorfizma. Za ilustracijo pogledimo vrsto iz prejšnjega primera. Če bi *PriorityQueue* bil objekt, definiran v objektno orientiranem jeziku, uporabniku ne bi bilo treba kreirati svoje vrste s klicem procedure *Create*, ampak bi enostavno deklariral objekt kot instanco razreda *Queue*. Še več: naš modul lahko kreira in obravnava poljubno število vrst, vendar so vse to vrste podatkov tipa *ProcId*; pri objektno orientiranem jeziku bi ista abstrakcija (razred) lahko obravnavala poljubno število vrst poljubnih elementov.

Procesna abstrakcija

Glavni izvor kompleksnosti sistemov realnega časa je v njihovi *sočasnosti*. Za obvladovanje te kompleksnosti je bistvena izrazna moč, ki jo daje *procesna abstrakcija*, s katero ponazarjamo operacije, ki morajo teči sočasno (paralelno). Procesna abstrakcija je glavni mehanizem dekomponiranja sistemov realnega časa. Brez konstruktov sočasnega programiranja bi bilo izredno težko narediti jasno preslikavo asinhronih entitet zunanjega sveta v program, posledica tega pa bi bilo težko zagotavljanje zanesljivosti programske opreme, še posebej v zvezi s časovnimi zahtevami. Zato so konstrukti sočasnega programiranja zelo važen element programskih jezikov realnega časa.

Sočasen program lahko definiramo kot program, v katerem paralelno (v večprocesorskem sistemu) ali navidezno paralelno (v enoprocessorskem sistemu) teče več sekvenčnih programov. Te programe imenujemo *proces*i. Navidezno paralelni procesi si medsebojno delijo procesor po določeni *strategiji razporejanja*. Proces i v sočasnem programu imajo *skupni cilj*, za doseganje katerega med sabo preko raznih mehanizmov *komunicirajo* in se *sinhronizirajo*. Proces i uporabljajo skupne *vire* (podatki, vhodno/izhodne enote, itd.), zato za te vire medsebojno *tekmujejo*, ta konflikt pa se rešuje s pomočjo raznih mehanizmov *medsebojnega izključevanja*. Medprocesna komunikacija, sinhronizacija in njihovo medsebojno izključevanje pri dostopu do skupnih virov predstavljajo probleme, ki sodijo med najzahtevnejše pri načrtovanju programske opreme nasploh. Te probleme rešujemo s pomočjo raznih mehanizmov, ki so del procesne abstrakcije, če pa niso, jih je potrebno realizirati s pomočjo obstoječih nižjenivojskih abstrakcij (konstruktov).

Del programa, ki ga več procesov mora izvesti medsebojno izključujoče, se imenuje *kritični odsek*. Najvažnejši mehanizmi medsebojnega izključevanja so *semaforji*, *monitorji* in *pošiljanje sporočil*.

Semafor je nenegativno celo število, ki ga procesi spreminjajo izključno preko klicev procedur *wait* (ob zajemanju skupnega vira) in *signal* (ob sproščanju skupnega vira). Pri tem je ključno to, da se ti dve proceduri morata izvajati nedeljivo ali *atomično*.

Čeprav semafor do določene mere rešuje problem medsebojnega izključevanja, je to še vedno zelo nizkonivojski konstrukt, katerega nepravilna uporaba lahko pripelje do nekaterih zelo hudih napak. Pri vseh teh napakah je značilno, da jih prevajalnik ne more odkriti in se zato pokažejo šele med izvajanjem programa. Celo pravilna uporaba

semaforjev praviloma pripelje do zamegljenih programov.

Pomanjkljivosti semaforjev v veliki meri odpravlja *monitor*. To je v bistvu *modul*, pri katerem samo en proces hkrati lahko izvaja modulsko proceduro. Z drugimi besedami je monitor višjenivojski mehanizem, ki zagotavlja hkrati strukturiran in medsebojno izključujoč dostop do skupnega vira.

Mehanizmi, ki temeljijo na pošiljanju sporočil, izhajajo iz distribuiranih sistemov. Od teh je najbolj znan *rendezvous*, ki ga srečamo pri programskem jeziku Ada.

V zvezi s procesno abstrakcijo lahko omenimo še nekaj lastnosti, ki naj bi jih imel programske jezik za sisteme realnega časa. To so definiranje časovnega obnašanja procesov (periodičnost, zakasnitve), postavljanje mejnih časov, izbira algoritma razvrščanja procesov in dinamične prioritete. Zaželenosti, ki pa jih zaenkrat ne podpira še noben jezik, pa so: evalvacija časa izvajanja, generiranje deterministične kode ter mehanizmi za analizo časovnega obnašanja. Te lastnosti so trenutno še vedno le predmet raziskav.

Nizkonivojsko programiranje

Računalniški sistem za delovanje v realnem času se od drugih računalniških sistemov razlikuje predvsem po načinu interakcije z okoljem. Tak sistem mora sinhronizirati svoje delovanje z zunanjimi dogodki ter komunicirati z različnimi nestandardnimi enotami. Zato pri takšnih sistemih ne moremo imeti standardnih knjižnic, ki bi omogočale komunikacijo na visokem nivoju z zunanjimi enotami, kot je to primer pri večini ostalih računalniških sistemov ali pri splošnih operacijskih sistemih. Zaradi tega pri sistemih realnega časa navadno ne najdemo več običajne strukture:

materialna oprema + operacijski sistem + aplikacijski program

ampak pogosto samo:

materialna oprema + aplikacijski program s komponentami operacijskega sistema

Iz tega sledi zahteva, da mora programske jezik omogočati programiranje na nizkem nivoju, v kar sodijo naslednje operacije:

- obravnavanje prekinitev;
- dostop do materialne opreme, npr. naslavljanje vhodno-izhodnih registrov;
- absolutno naslavljanje;
- dostop do posameznih bitov in njihovo manipuliranje;
- vstavljanje kode v zbirnem jeziku;
- razširitev jezika s posebnimi instrukcijami značilnimi za posamezne procesorje.

Pri nizkonivojskem programiranju prihaja do opuščanja pravil konsistentnosti tipov, kar zmanjšuje zanesljivost. Zato mora biti nizkonivojsko programiranje ločeno od ostalega dela programa.

Obravnavanje izjemnih situacij med izvajanjem

Računalniški sistem realnega časa ne sme obstati, če med izvajanjem pride do kakšne izjemne situacije, ampak mora nadaljevati delovanje. Izjemno situacijo (napako) lahko odkrije:

1. implementacijski nivo, to je materialna oprema (npr. deljenje z 0) ali koda, ki jo vrine prevajalnik (npr. prekoračitev polja). Te napake so hujše, saj lahko nastopijo kjerkoli, tudi sredi izraza;
2. izrecno programirani testi stanja sistema (npr. *IF NOT pogoji THEN napaka*).

Pri računalniških sistemih, ki niso namenjeni vodenju v realnem času, je pogosto ustrezen odgovor na napako tudi sporočilo o napaki in prekinitev izvajanja programa (*fail-hard*), sistem za vodenje v realnem času pa mora biti zmožen omejiti posledice napake (*fail-safe*) in/ali nadaljevati z delovanjem, čeprav mogoče z zmanjšano funkcionalnostjo (*fail-soft*).

Zahteve, ki jim mora zadoščati mehanizem za obravnavanje izjemnih situacij, so naslednje:

- biti mora enostaven;
- količina dodatne kode mora biti čim manjša;
- dodatna koda se sme izvajati samo v primeru nastopa izjemne situacije;
- vse napake mora obravnavati enako, ne glede na izvor in mehanizem odkrivanja napake.

8.3.3.3 Kratek pregled jezikov

V sistemih realnega časa se je v preteklosti uporabljalo veliko število programskih jezikov. Tako na primer obstaja ocena, da so v sklopu projektov za ameriško obrambno ministrstvo v sedemdesetih letih uporabljali več kot tristo programskih jezikov (to je bil povod za razvoj jezika Ada).

Če se omejimo samo na nekaj najbolj znanih, lahko omenimo naslednje jezike: Ada, BASIC, C, C++, Coral66, Forth, FORTRAN, Modula-2, Modula-3, Oberon/Oberon-2, Pascal, Pearl, Occam, RTL-2, Parallel C, Concurrent Pascal, Smalltalk in Conic. Nekateri od teh jezikov so bili uporabljani samo v zelo ozkem krogu, od tistih, ki so nekoč imeli širši krog uporabnikov, pa nekatere izpodrivajo drugi. Zato posebej izdvojimo naslednje jezike: Ada, C, C++, Modula-2, Oberon-2.

Glede na njihove lastnosti in seveda ob upoštevanju trenutnega stanja, standardov, dostopnosti prevajalnikov, razvojnih okolij in knjižnic, je verjetno, da bo v prihodnje stanje na tem področju sledeče (Cooling, 1996):

- Ada bo verjetno prevladovala v kritičnih aplikacijah, zaradi svoje visoke zanesljivosti in varnosti.

- C bo verjetno ostal dominanten jezik za vložene (embedded) aplikacije, predvsem zaradi tradicije in velike razširjenosti.
- C++ bo postal najvažnejši jezik na področju delovnih postaj in, v nekoliko manjši meri, na PC platformah. Visual C++ se bo največ uporabljal za programiranje uporabniških vmesnikov (GUI). Njegov prihod v področje vložnih aplikacij preprečuje njegova prevelika kompleksnost.
- Modula-2 ima perspektivo pri kritičnih aplikacijah zaradi svoje zanesljivosti. Prisotna bo verjetno tudi pri vložnih aplikacijah.
- Oberon-2 bo verjetno postal dominanten jezik na področju raziskav in izobraževanja, v nekaterih okoljih pa bi prav lahko postal zamenjava za C++ ali za Ado. Obstaja tudi velika verjetnost njegove uporabe v kritičnih sistemih.

V sistemih realnega časa se v zadnjem času pogosto uporablja kombinacija dveh jezikov: za kritične dele aplikacij se uporablja Ada, ostali deli pa se programirajo v C++.

Z naraščanjem deleža stroškov razvoja in posebej vzdrževanja programske opreme sistemov realnega časa, se vedno bolj kaže potreba po uporabi visokonivojskih strukturiranih programskih jezikov s strogo zahtevo enakosti tipov, sofisticiranimi mehanizmi abstrakcije ter mehanizmi sočasnega programiranja. Ta potreba je sorazmerna z velikostjo sistema, velikostjo razvojne skupine ter verjetnostjo pogostega spreminjanja ali dopolnjevanja programske opreme.

8.4 Orodja za konfiguriranje in implementacijo funkcij vodenja

V začetnem delu tega poglavja smo govorili predvsem o orodjih, ki so bolj splošnonamenska. To po eni strani pomeni, da niso vezana samo na problemsko domeno vodenja procesov, ampak so uporabna tudi na drugih področjih. Po drugi strani pa to seveda pomeni, da so precej fleksibilna in univerzalna, zato omogočajo realizacijo kakršnekoli funkcije vodenja.

V nasprotju s tem bomo v tem podpoglavju govorili o orodjih, ki so bolj specifična oziroma bolj problemsko usmerjena v vodenje. Njihova specifičnost se kaže bodisi v specifičnem naboru funkcij, katerih realizacijo omogočajo, ali pa specifični računalniški materialni opremi, na kateri se izvajajo. Ker imajo določen omejen namen, so seveda bolj učinkovita in praviloma tudi manj fleksibilna.

V tem delu bomo dali poudarek štirim vrstam takih orodij. Najprej bomo govorili o orodjih, ki omogočajo realizacijo ekspertnih sistemov za vodenje in so uporabna predvsem za zahtevnejše funkcije nadzora. Nato bomo predstavili tako imenovana SCADA orodja, ki so v glavnem namenjena za realizacijo osnovnih funkcij nadzora kot so prikaz podatkov, protokoliranje, arhiviranje, komunikacija z operaterjem, itd. Tako

prva kot druga orodja tečejo na računalnikih, ki so pretežno namenjeni nadzoru procesov. Za razliko od tega, orodja za krmilnike in regulatorje, ki jih bomo predstavili v nadaljevanju podpirajo konfiguriranje oz. programiranje krmilnikov in regulatorjev, ki so bodisi samostojni izvajalci funkcij vodenja ali pa predstavljajo del v distribuiranem sistemu.

Kot seveda sledi že iz imen so programi za krmilnike pretežno (ne pa v celoti) namenjeni realizaciji funkcij krmiljenja, programi za regulatorje pa pretežno (ne pa v celoti) realizaciji funkcij regulacije.

8.4.1 Orodja za razvoj ekspertnih sistemov vodenja

Kot smo omenili že v podpoglavju 7.4 zanimanje za metode umetne inteligence, predvsem ekspertne sisteme na področju vodenja, temelji na dejstvu, da klasične metode niso primerne za reševanje celotnega spektra problemov avtomatizacije, predvsem na področjih, kjer človek s svojimi izkušnjami in sposobnostjo kvalitativnega sklepanja še vedno igra pomembno vlogo (Mc. Ghee in sodel, 1990). Ta ugotovitev velja predvsem za višje nivoje vodenja, kjer gre, še posebej pri razvoju nadzornih sistemov, za tesno prepletanje analitičnega in hevrističnega znanja. Prednosti ekspertnih sistemov pred klasičnimi orodji, ki se uporabljajo za reševanje problemov vodenja na vseh področjih (npr. SCADA sistemi za razvoj nadzornih sistemov), se kažejo predvsem v moči izraznosti jezika, ki poleg analitičnih rešitev omogoča tudi vključevanje znanja, ki ga je mogoče opisati s pomočjo kvalitativnih pravil in izrazov.

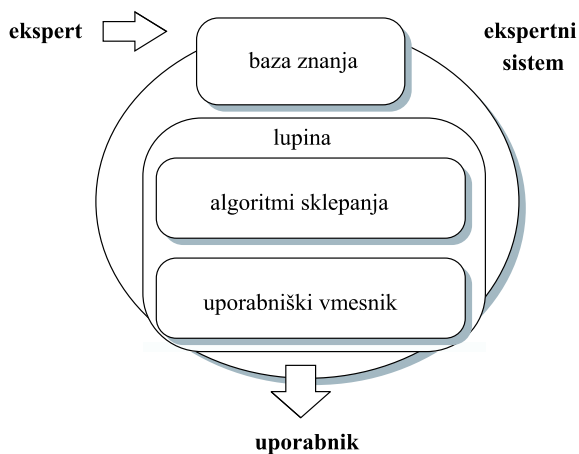
8.4.1.1 Ekspertne lupine

Zaradi naraščajočega zanimanja za ekspertne sisteme (ES) so se na tržišču začela pojavljati orodja, ki podpirajo razvoj ekspertnih sistemov. Taka orodja (t.i. *ekspertne lupine*) vsebujejo algoritme za procesiranje znanja (t.i. *mehanizem sklepanja*) in uporabniški vmesnik ter omogočajo razvijalcu, da se osredotoči le na razvoj baze znanja (Sl. 8.26).

Baza znanja predstavlja najpomembnejši del ekspertnega sistema in je odvisna od konkretne aplikacije.

Mehanizem sklepanja je tisti modul, ki omogoča aktivno uporabo znanja v bazi preko algoritmov sklepanja in s tem reševanje problemov. Deluje kot inteligentni interpreter pravil v bazi znanja. Na osnovi znanja, ki je shranjeno v bazi, mora poiskati najustreznejšo rešitev s pomočjo principov sklepanja.

Zelo pomembna je tudi možnost komunikacije uporabnika s sistemom. To je naloga *uporabniškega vmesnika*.



Sl. 8.26. Zgradba ekspertnega sistema

Ekspertne lupine vsebujejo mehanizme sklepanja, ki podpirajo izbrane formalizme predstavitve znanja, modul za pojasnjevanje in uporabniški vmesnik. Ponavadi so razvite v enem izmed visoko-nivojskih programskih jezikov - Prolog, Lisp, Pascal ali C. Ekspertne lupine se razlikujejo glede na:

- vrsto formalizmov za predstavitev znanja v bazi;
- postopke sklepanja;
- dovoljeno kompleksnost baze znanja;
- razvojno okolje in grafično podporo.

Razvojno gledano lahko orodja za podporo razvoja ekspertnih sistemov razdelimo v dve skupini.

Ekspertne lupine 1.generacije

V zgodnjem obdobju so ekspertni sistemi nastopali v funkciji t.i. ekspertne pomoči uporabniku. Omogočali so reševanje problemov na osnovi ekspertnega znanja in so bili sposobni svojo rešitev tudi pojasniti (Jackson, 1990, Carrico in sodel., 1989). Kot posledica tega dejstva, večina tovrstnih ekspertnih lupin omogoča sklepanje izključno na osnovi hevrističnega ali kvalitativnega znanja. Zato z njimi lahko uporabljamo formalizme, ki podpirajo opis kvalitativnih izrazov. Druga značilnost teh ekspertnih lupin je, da se odzivajo na zahteve uporabnika in ne podpirajo arhitektur za delo v realnem času. Omejitve se kažejo tudi pri zagotavljanju komunikacije z zunanjimi sistemi in podatkovnimi bazami. Nekatere ekspertne lupine, ki sodijo v to skupino so Level5 Object (Information Builders Inc), Kappa PC (Intellicorp), itd.

Ekspertne lupine 2.generacije

Prve komercialno dostopne ekspertne lupine, ki so temeljile na izhodiščih t.i. ekspertnih

sistemov prve generacije, zaradi pretežne omejenosti na opis kvalitativnih izrazov v praksi niso bile uspešne. Zaradi vedno večjih zahtev po zanesljivosti in celovitosti industrijskih sistemov vodenja se je izrazito pojavila potreba po uporabi orodij z arhitekturami, namenjenimi za delo v realnem času, bogato izraznostjo jezika in močnimi uporabniškimi vmesniki. Naštete zahteve so bile kasneje upoštevane pri razvoju ekspertnih lupin 2. generacije (Arzen, 1992, Boullart in sodel., 1992, Vaesen, 1992).

Zanje so značilni predvsem mehanizmi za podporo procesiranja znanja v realnem času, možnost integracije različnih konceptov predstavitve znanja, ki omogočajo opis hevrističnega in analitičnega znanja o procesu. Pomembni so tudi integrirani vmesniki za dostop do podatkovnih baz, realnih sistemov vodenja (npr. tehnološki procesi) in ostalih podatkovnih virov.

Filozofija, prisotna pri razvoju sodobnih ekspertnih lupin 2. generacije, je osnovana na ideji integriranega okolja, v katerem so združeni koncepti razvoja ekspertnih sistemov, klasičnega programiranja in objektne tehnologije. Orodje lahko opišemo kot okolje z dobro razvitim uporabniškim vmesnikom, bogato izraznostjo jezika in arhitekturami, namenjenimi za delo v realnem času. Poleg tega sledi tudi usmeritvam razvoja porazdeljenih mrežnih sistemov. Proizvajalce, ki razvijajo ekspertne lupine 2. generacije lahko razdelimo v naslednji skupini:

- podjetja, ki razvijajo programsko opremo in sledijo trendom s področja umetne inteligence. V to skupino orodij sodijo ekspertne lupine RTAC (Mitech Co.), CogSys (Cogsys Ltd), RTWorks (Talarian Corporation), G2 (GenSym Co.) in druga;
- podjetja s področja razvoja sistemov vodenja, kot so Honeywell, Hitachi, Toshiba, ki standardnim sistemom vodenja dodajajo ekspertne lupine, s katerimi je mogoče razviti ekspertne module in jih integrirati v standarne sisteme vodenja. Eno takih orodij je TDC 3000 Expert (Honeywell).

8.4.1.2 Koncepti gradnje baze znanja v ekspertnih lupinah

Baza znanja vsebuje znanje o problemski domeni. Sem sodijo enostavna *dejstva o problemu, pravila, ki opisujejo relacije med objekti v domeni ter ideje in postopki za reševanje problemov v domeni*. Naloga tehnologa znanja (angl. knowledge engineer) je, da s pomočjo strokovne literature in pogovorov s strokovnjaki zbere znanje o problemski domeni in ga predstavi v računalniku razumljivi obliki. Pri tem se postavljata dve vprašanji:

- v kakšni obliki predstaviti znanje, in
- kako zgraditi bazo znanja.

▽

Primer 8.8: Baza znanja za odkrivanje napak

Primer baze znanja za odkrivanje napak realizirane na osnovi pravil v orodju G2 je prikazan na Sl. 8.27.

Faults - Syptoms Table From Fault Trees

	C	UNIT	FAULT	CL0	H	Q	S	
1		V1	"clog"	LOW	HIGH	LOW	LOW	
2		V1	"clog"			LOW	LOW	
3		V2	"clog"	LOW	HIGH	HIGH	LOW	
4		V2	"clog"			HIGH	LOW	
5		Div	"clog out 2 "	LOW	HIGH	HIGH	LOW	
6		Div	"clog out 2 "			HIGH	LOW	
7		R1	"clog outlet"	LOW	HIGH	LOW	LOW	
8		R1	"clog outlet"			LOW	LOW	
9		LT1	"sensor-fail (on hig.	LOW	HIGH	LOW	LOW	
10		LT1	"sensor-fail (on hig.			LOW	LOW	
11		V1	"leak"	HIGH	LOW	HIGH	HIGH	
12		V1	"leak"			HIGH	HIGH	
13		V2	"leak"	HIGH	LOW	LOW	HIGH	
14		V2	"leak"			LOW	HIGH	
15		P1	"clog"	HIGH	LOW	LOW	HIGH	

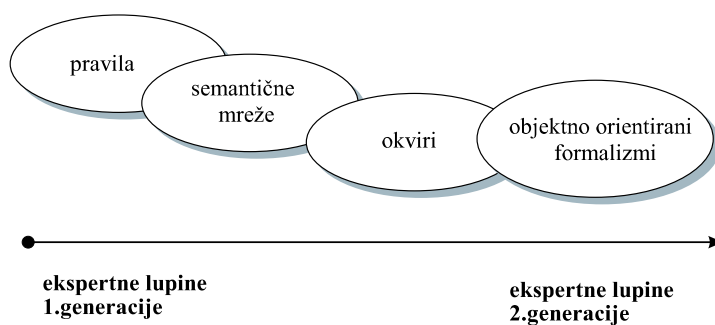
OK

Sl. 8.27. Baza znanja za odkrivanje napak realizirana na osnovi pravil.

Δ

Formalizmi za predstavitev znanja

Način predstavitve znanja v ekspertni lupini je odvisen od vrste formalizmov, ki jih tako orodje podpira. Med najbolj pogosto uporabljene formalizme sodijo (Sl. 8.28):



Sl. 8.28. Formalizmi za predstavitev znanja.

- pravila,
- semantične mreže,
- okviri in
- objektno orientirani formalizmi.

Pravila

Pravila so najbolj razširjen formalizem, ki se za predstavitev znanja uporablja predvsem v

ekspertnih lupinah 1.generacije. Ta formalizem dokaj dobro omogoča predstavitev kvalitativnega znanja. Glavne pomankljivosti pa se izražajo predvsem pri opisu analitičnega znanja.

Najbolj pogosta so pravila oblike **ČE** - **POTEM** (angl. IF-THEN) , ki imajo lahko različne interpretacije:

ČE < Pogoji P > **POTEM** < Zaključek C >

ČE < Situacija S > **POTEM** < Akcija A >

Poleg tovrstnih pravil se uporabljajo tudi **WHEN** ali **WHENEVER** pravila, brezpogojna pravila, inicializacijska pravila, itd.

▽

Primer 8.9: Realizacija nekaterih tipov pravil

Nekateri tipi pravil realizirani v okolju G2 so prikazani na Sl. 8.29.

initially conclude that LT1 = 10	
whenever LT1 receives a value then start my-procedure (the value of LT1, my-array)	
when LT1 < 1000 then change the text of the formula of Q3 to "LT1 * 3.24e2"	
unconditionally conclude that LT1 = LT1 + 10	
if LT1 > 1000 then change the text of the formula of Q3 to "none"	
a rule	
Options	invocable via backward chaining, invocable via forward chaining, may cause data seeking, may cause forward chaining
Notes	OK
Authors	mateja (18 May 1997 3:07 p.m.)
Item configuration	none
Names	none
Tracing and breakpoints	default
if LT1 > 1000 then change the text of the formula of Q3 to "none"	
Scan interval	none
Focal classes	none
Focal objects	none
Categories	none
Rule priority	6
Depth first backward chaining precedence	none
Timeout for rule completion	base default

Sl. 8.29. Tipi pravil realizirani v orodju G2.

△

Okviri

Okviri sodijo, poleg pravil, med tradicionalne formalizme predstavitev znanja, ki jih najdemo v ekspertnih lupinah 1. generacije. Okvir je struktura, ki se uporablja za opis lastnosti, t.j. atributov objektov predstavljenih v bazi znanja (Jackson, 1991). V poljih okvirja se nahajajo atributi objektov, ki jim lahko pripišemo pripadajočo vrednost, pravilo ali postopek, na katerega se nanaša.

Objektna predstavitev znanja

Tradicionalni formalizmi za predstavitev znanja niso dovolj zmogljivi in fleksibilni, da bi lahko sledili paradigmi ekspertnih sistemov 2. generacije. Zaradi tega se v sodobnejših ekspertnih lupinah uporabljajo, poleg ostalih formalizmov, tudi objektno orientirani formalizmi predstavitev znanja, ki smo jih predstavili že v prejšnjem poglavju. Tovrstni formalizmi v razvoju lupin nadomeščajo strukturo okvirja, ki je objektno zasnovana, a po izrazni moči ne dosega pravih objektno orientiranih formalizmov.

Bazo znanja ekspertnega sistema opisujemo s pomočjo objektov. Vsak objekt je razpoznaven z množico *atributov* (internih spremenljivk objekta) in množico *metod*, ki operirajo nad objektovimi atributi. Operacije nad objekti opišemo z uporabo različnih tipov pravil, klasičnih procedur ali objektu pripadajočih metod.

Objekti, ki imajo podobne lastnosti se združujejo v razrede. Razred torej predstavlja osnovni vzorec množice podobnih si objektov, kjer se podobnost izraža v enaki množici atributov in metod. Nekatera programska orodja (npr. G2) podpirajo vizualno programiranje in omogočajo, da je vsak razred razpoznaven preko pripadajoče grafične podobe.

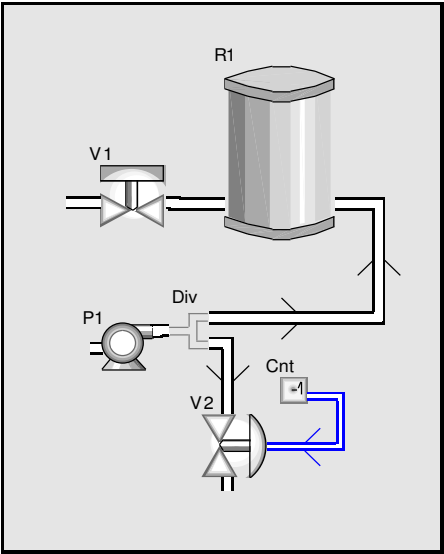
▽

Primer 8.10: Definicija in grafični prikaz objekta

Na Sl. 8.30 vidimo prikaz definicije objekta, na Sl. 8.31 pa so grafično prikazani objekti, t.j. procesni elementi, ki so značilni za razvoj ekspertnih sistemov s področja vodenja. Oboje je narejeno z orodjem G2.

 VESSEL-22	VESSEL-22, an object-definition	
	Notes	OK
	Authors	none
	Item configuration	none
	Class name	vessel-22
	Direct superior classes	bsc-unit
	Class specific attributes	none
	Instance configuration	none
	Change	none
	Menu option	a final menu choice
	Class inheritance path	vessel-22, bsc-unit, bsc-object, object, item
	Inherited attributes	id-unit-name is a text, initially is "NO NAME"; id-unit is an integer, initially is 0; description is a text, initially is "Description for the component does not exist."
	Attribute initializations	description initially is "Special type of valve"; id-unit-name initially is "SVx"
	Attribute displays	inherited
	Stubs	an input con_1 in1 located at left 5; an input con_1 in2 located at left 25; an output con_1 out1 located at right 5; an output con_1 out2 located at right 25
	Icon description	width 63; height 34; light-gray: filled rectangle (0, 0) (63, 34); light-steel-blue: filled rectangle (2, 2) (62, 32); black: outline (0, 0) (0, 34) (63, 34) (63, 0)

Sl. 8.30. Prikaz definicije objekta v orodju G2.



Sl. 8.31. Grafični prikaz objektov.

Razredi so organizirani v hierahično strukturo, v kateri velja pravilo enojnega ali večkratnega *dedovanja*. Dedovanje omogoča, da se lastnosti (atributi in metode) nadrejenega razreda prenesejo na njegovega naslednika. V primeru večkratnega dedovanja lahko razred podeduje lastnosti vseh razredov, iz katerih izhaja. Poleg principa dedovanja sta prisotna tudi principa *enkapsulacije* (princip zapiranja podatkov in operacij) in *polimorfizma* (imensko enake operacije se lahko obnašajo različno v različnih razredih).

Kako zgradimo bazo znanja

Med ekspertnimi lupinami prevladujejo take, v katerih je potrebno bazo znanja zgraditi "ročno" - s pomočjo strokovne literature in pogovorov s strokovnjaki. To je praviloma zelo dolgotrajen proces, saj svoj "know-how" strokovnjak ponavadi težko pretvori v "say-how" tako sistematično in podrobno, kot jo zahtevajo postopki predstavitve znanja v računalniku. Zato so poznejšim generacijam ekspertnih lupin pridružena orodja, ki omogočajo podporo pri sistematični graditvi baze znanja (npr. XpertRule, Attar Software Limited).

8.4.1.3 Mehanizmi sklepanja

Poleg problema predstavitve znanja je bistvenega pomena vprašanje, kako znanje učinkovito uporabiti oz. kako poiskati odgovor na zastavljeno vprašanje uporabnika in pojasniti odgovor. Odgovor na uporabnikovo vprašanje je včasih možno dobiti že s samo uporabo dejstev v bazi. V nasprotnem primeru pa mora ekspertni sistem uporabiti sklepanje. Računalniški sistem mora izpeljati in preveriti dejstva, ki mu niso eksplicitno podana. Procesiranje baze znanja opravlja poseben mehanizem sklepanja (angl. *Inference Engine*).

Prve ekspertne lupine (npr. Level5 Object, Expro, Insight 1, itd.) so podpirale predvsem dva mehanizma sklepanja:

- *sklepanje naprej* (angl. forward chaining)
Poteka od danih dejstev k hipotezi. Iz znanih dejstev generiramo nova, dokler ne pridemo do dejstva, ki se sklada s ciljno hipotezo.
- *sklepanje nazaj* (angl. backward chaining)
Se izvaja v obratni smeri - od ciljne hipoteze k znanim dejstvom, ki so v bazi znanja ali za katere sistem vpraša uporabnika. Iz ciljne hipoteze generiramo nove hipoteze, dokler ne pridemo do hipotez, ki so enaka znanim dejstvom.

Poseganje na področje vodenja z ekspertnimi lupinami 2.generacije (npr. G2, CogSys, itd.) pa zahteva mehanizme, ki omogočajo sklepanje v realnem času in generirajo odločitve v zahtevanih časovnih okvirih. Časovna zakasnitev med sprejetimi vhodnimi podatki in kontrolno akcijo je odvisna od števila pravil in hitrosti algoritmov sklepanja. Ta problem se rešuje z uporabo konceptov progresivnega sklepanja (Boullart in sodel., 1992). Značilnosti tovrstnih mehanizmov so, da izvajajo sklepanje v vnaprej določenih

časovnih intervalih ali asinhrono, ob pojavu določenega dogodka. Pomemben parameter teh sistemov je čas in mehanizmi, ki omogočajo sklepanje na osnovi časovnega zaporedja podatkov ali dogodkov.

Zaradi zagotavljanja širše uporabe sodobne ekspertne lupine (npr. G2) omogočajo izbiro načina izvajanja sklepanja:

- v realnem času;
- v simulacijskem času in
- v najkrajšem možnem času (angl. "as fast as possible"). Pri uporabi tega načina se nov cikel izvajanja pravil in procedur začne takoj, ko se vsa pravila v prejšnjem ciklu izvršijo.

8.4.1.4 Vmesniki ekspertnih lupin

Ekspertni sistemi 1. generacije so služili kot ekspertna pomoč uporabniku, zato je bila komunikacija uporabnika s sistemom dobro podprta v prvih ekspertnih lupinah. Interakcija z uporabnikom mora biti "prijazna" - enostavna, razumljiva, omogočati mora tudi pojasnitev rešitve. Večina ekspertnih lupin vključuje grafično podporo za interakcijo z uporabnikom.

Ekspertne lupine za delo v realnem času pa poleg uporabniškega vmesnika vsebujejo tudi vmesnike za dostop do podatkovnih baz, podatkov iz realnih procesov in ostalih podatkovnih virov.

Poleg ostalih že naštetih modulov, sodobna orodja za razvoj ekspertnih sistemov vsebujejo tudi gradnike, ki podpirajo dinamično modeliranje in simuliranje procesov (npr. G2). Vgrajeni simulator in povezljivi grafični objekti omogočajo hitro modeliranje in simuliranje zgrajenega modela. Simulator je posebne vrste podatkovni strežnik, ki zagotavlja simulirane vrednosti za spremenljivke in parametre.

8.4.2 Programska orodja za računalniške nadzorne sisteme (SCADA)

O nadzoru kot eni najbolj razširjenih funkcij vodenja smo obširneje spregovorili že v sedmem poglavju. Na tem mestu se bomo osredotočili na programska orodja, ki pokrivajo večino enostavnejših funkcij nadzora kot so npr.: zbiranje podatkov, spremljanje, prikaz in delovanje procesa, alarmiranje, arhiviranje podatkov in generiranje poročil.

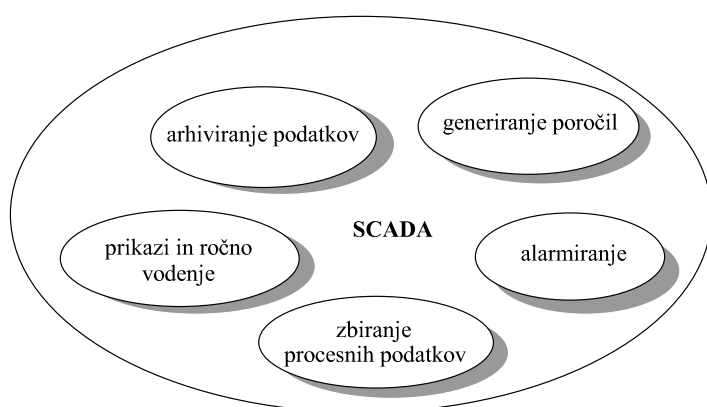
Programska orodja, ki podpirajo razvoj in izvedbo aplikacij na nadzornih računalniških sistemih se imenujejo tudi orodja SCADA (angl. Supervisory Control And Data Acquisition). Programska orodja tipa SCADA so se najprej pojavila na centraliziranih sistemih vodenja, na velikih računalnikih. Uporabljali so jih v elektrarnah, dispečerskih centrih in drugih večjih sistemih. S padanjem cen računalnikov in rastjo njihove

zmogljivosti je prišlo tudi do njihove širše uporabe. Pravi razmah so dosegli z uvedbo porazdeljenih sistemov vodenja, osebnih računalnikov ter dostopnih cen delovnih postaj in miniračunalnikov. Z razvojem lokalnih omrežij in uporabo sodobnih orodij tipa SCADA lahko omogočimo vpogled v delovanje procesa in zbirko procesnih podatkov tudi strokovnjaku na oddaljenem delovnem mestu.

8.4.2.1 Osnovne funkcije orodij SCADA

Programska orodja SCADA omogočajo razvoj nadzornih sistemov z naslednjimi funkcijami (Sl. 8.32):

- zbiranje procesnih podatkov;
- prikazi in ročno vodenje;
- alarmiranje;
- arhiviranje/shranjevanje procesnih podatkov ter
- generiranje poročil.



Sl. 8.32. Osnovne funkcije orodij SCADA

Zbiranje procesnih podatkov

Celovito obladovanje industrijskih procesov vodenja zahteva sprotno spremljanje procesa in posredovanje v realnem času. Tovrstne zahteve vplivajo na razvoj orodij SCADA, ki vsebujejo

- arhitekture za izvajanje v realnem času in
- vmesnike za dostop do procesnih podatkov.

Sistemi SCADA omogočajo branje in pisanje podatkov iz procesnega računalnika, prikazovanje na ekranu ter shranjevanje na disk v vnaprej zahtevanem času, ki je odvisen od narave procesa. Zbiranje podatkov se lahko izvaja v določenem časovnem intervalu ali ob pojavu določenega dogodka t.j. dogodkovno. Poleg tega je hitrost zbiranja podatkov odvisna tudi od naslednjih faktorjev: zmogljivosti nadzornega

računalnika, vrste operacijskega sistema, načina povezave s procesnim računalnikom, organizacije podatkovne baze v hitrem spominu ter hitrosti izvajanja posameznih programskih modulov.

Splošno pravilo je, da je hitrost manipuliranja s procesnimi podatki odvisna od vrste in zahtev procesa, ki ga vodimo. Izbira daljšega intervala zbiranja podatkov je ustrezna za nadzor počasnih procesov, v katerih se spremembe dogajajo na daljši časovni interval.

Za realizacijo funkcije zbiranja procesnih podatkov z orodji SCADA potrebujemo vmesnike za dostop do procesnih podatkov. Nekateri standardni procesni vmesniki so dostopni v paketu z orodjem SCADA, druge pa je potrebno dobiti od proizvajalcev procesnih računalnikov, krmilnikov, reaktorjev ali drugih računalniških naprav, preko katerih zajemanje podatkov poteka.

Prikazi in ročno vodenje

Poleg sprotnega zbiranja procesnih podatkov so sistemi SCADA tudi v funkciji podpore operaterju in omogočajo vpogled in poseganje v delovanje procesa. Interakcija z operaterjem mora biti enostavna in razumljiva. Pomemben element uporabniških vmesnikov so prikazi, ki so podani bodisi v grafični ali tekstovni obliki, stanje procesa pa prikazano s pomočjo animacije posameznih procesnih komponent. Sodobni SCADA sistemi vsebujejo gradnike, ki podpirajo vizualno programiranje, razvoj grafičnih vmesnikov in animiranih elementov. Tem zahtevam sledi tudi uporaba paradigme objektno-orientiranih pristopov v sistemih SCADA.

Alarmiranje

Pomembna funkcija nadzornih sistemov je tudi obveščanje operaterja (t.j. alarmiranje) v primeru odstopanj merljivih veličin iz območja dovoljenih vrednosti. Orodja SCADA podpirajo definiranje seznama alarmov in pogoje za nastop le-teh. Glede na vrsto ali kritičnost pojava alarma je mogoče alarme grupirati v posamezne skupine (npr. alarmi, opozorila, obvestila, itd). V primeru pojava posameznih alarmov pa sledi obvestilo operaterju z opisom vrste alarma.

Arhiviranje podatkov

Poleg sprotnega spremljanja delovanja procesa je pomembna funkcija nadzornih sistemov tudi shranjevanje in arhiviranje podatkov. Večina orodij SCADA dostopnih na tržišču omogoča shranjevanje v obliki podatkovne baze. Pri tem se sodobna orodja ne omejujejo le na določeno vrsto podatkovne baze temveč zagotavljajo pokritost najbolj razširjenih, npr. dBase, Microsoft Access, SQL server, Sybase, Oracle.

Generiranje poročil

Sistemi SCADA poleg že omenjenih funkcij omogočajo tudi generiranje različnih izpisov in poročil na osnovi tekočih procesnih podatkov.

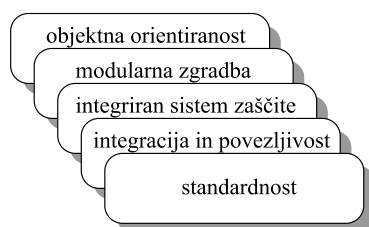
8.4.2.2 Lastnosti orodij SCADA

Lastnosti orodij SCADA lahko opišemo z dveh vidikov:

- z vidika razvijalca aplikacije, katerega zahteve so povezane z lastnostimi učinkovitega in uporabniku prijaznega razvojnega okolja ter
- z vidika uporabnika, t.j. operaterja, katerega zahteve se nanašajo predvsem na funkcionalnost in interakcijo z aplikacijo nadzornega sistema.

Pri sodobnih orodjih SCADA lahko govorimo o upoštevanju naslednjih lastnosti (Sl. 8.33):

- uporaba objektno-orientirane paradigme;
- modularna zgradba;
- integriran sistem varnosti in zaščite;
- povezljivost in odprtost;
- standardnost.



Sl. 8.33. Lastnosti SCADA orodij

Paradigma objektno-orientiranih pristopov

Osnovni gradniki aplikacije razvite z orodjem SCADA so objekti. Vsak objekt je razpoznaven z množico *atributov* in množico *metod*, ki operirajo nad objektovimi atributi. K definiciji objekta sodijo tudi atributi in metode, s katerimi definiramo način animacije določenega objekta.

Objekti, ki imajo podobne lastnosti se združujejo v razrede. Razred torej predstavlja osnovni vzorec množice podobnih si objektov, kjer se podobnost izraža v enaki množici atributov in metod. Vsak objekt tako predstavlja pojavitev določenega razreda. Če orodje podpira vizualno programiranje, je vsak razred razpoznaven tudi preko pripadajoče grafične podobe.

Modularna zgradba

Programska orodja SCADA so zasnovana modularno. Posamezne programske module lahko konfiguriramo v skladu z zahtevami aplikacije nadzornega sistema. Optimalna modularna zgradba je taka, pri kateri so okoli jedra, ki je običajno baza podatkov,

nanizani med seboj neodvisni programski moduli. Da je to zagotovljeno, posamezni programski moduli komunicirajo posredno preko jedra orodja SCADA.

Integriran sistem zaščite

Orodja SCADA zagotavljajo razvoj aplikacije z vgrajenim sistemom zaščite, ki zagotavlja kontrolo dostopa do posameznih funkcij le pooblaščenim osebam. Ta lastnost omogoča definiranje razredov uporabnikov s pripadajočimi pristojnostimi.

Integracija in povezljivost

Razvoj distribuiranih sistemov vodenja postavlja zahteve po gradnikih, ki omogočajo integracijo in povezljivost nadzornih sistemov.

Odrprtost pomeni, da je aplikacijo razvito z orodjem SCADA mogoče integrirati z drugimi programskimi orodji (npr. orodja Excel, Visual Basic ali Access) znotraj operacijskega sistema na standarden način z uporabo protokolov DDE (Dynamic Data Exchange), OLE (Object Linking and Embedding, OPC (OLE for Process Control), ODBC (Open Data Base Connectivity) in drugih.

Razvoj lokalnih omrežij pogojuje tudi zahteve po uporabi nadzornih sistemov na oddaljeni lokaciji, predvsem s ciljem sledenja procesa in prenosa procesnih podatkov za nadaljnjo obdelavo. V ta namen mora orodje SCADA vsebovati vmesnik za priključitev na omrežje in možnost pregleda in spreminjanja podatkov s pomočjo brkljalnikov prek danes vse bolj popularnega Internet omrežja (uporaba ActiveX protokola).

Standardnost

Standardnost orodja SCADA je zagotovljena s podporo standardnih in že uveljavljenih rešitev ter orodij, npr. podpora standardnih protokolov, programskih orodij ali podatkovnih baz.

8.4.2.3 Predstavitev konkretnega orodja SCADA in primer uporabe

Ponudba programskih orodij tipa SCADA je na svetovnem trgu zelo široka in raznovrstna. Pregled SCADA orodij je podan v posebni izdaji revije Control Engineering (Morris, 1998). Proizvajalce orodij SCADA lahko razdelimo v več skupin:

- proizvajalci procesne opreme, ki zagotavljajo integrirane rešitve računalniške podpore procesnega in nadzornega nivoja. Zato so razvili tudi lastna orodja tipa SCADA. V to skupino sodijo orodja RSVIEW (Rockwell), LabView (National Instruments), Coros (Siemens), itd.;
- neodvisni proizvajalci orodij SCADA (ki zagotavljajo povezljivost z določenimi proizvajalci); v tej skupini so orodja FactoryLink (USData), FIX (Intellution), InTouch (Wonderware), Wizcon (PC Soft), Genesis (Iconics), itd.

Nekoliko podrobneje bomo predstavili programsko orodje Factory Link USData.

Njegove glavne značilnosti bomo podali v nadaljevanju.

Objektno programiranje

Osnovni elementi vsake aplikacije so objekti, katerim pripadajo atributi in metode, ki opisujejo operacije nad objektom. Z uporabo atributov in metod definiramo tudi animacijo objekta. Enostavne objekte je mogoče sestaviti v kompleksnejše objekte tako, da vgrajeni objekti obdržijo vse svoje lastnosti. Za opis objektov je poleg običajnih grafičnih in vpisnih metod, na voljo tudi PowerVB, kar je prireditev Visual Basica za uporabo znotraj programskega paketa FactoryLink.

Proženje na dogodke

Programski moduli delujejo na osnovi dogodkov. V primeru pojava določenega dogodka se odzovejo le programski moduli, ki čakajo na ta dogodek. Na ta način lahko dosežemo razbitje aplikacije na posamezne module, ki delujejo le kratek čas. S tem so doseženi zelo hitri odzivni časi ter zmanjšan promet na povezavah.

Shranjevanje podatkov

Factory Link podpira uporabo vrste najbolj razširjenih podatkovnih baz, t.j. dBase, Access, SQL server, Sybase, Oracle in druge. Pri generiranju poročil pa se podatki shranjujejo v tekstovni ASCII obliki.

Standardne povezave na ravni operacijskega sistema

Orodje FactoryLink omogoča DDE, OLE, ActiveX object povezave znotraj Windows operacijskega sistema.

Vizualizacija aplikacije na daljavo

WebClient tehnologija orodja FactoryLink omogoča s pomočjo Internet/Intranet povezav in ActiveX tehnologije, spremembo Windows NT operacijskega sistema iz enouporabniškega v večuporabniški operacijski sistem. S tem omogoča prenos pregleda prikazov aplikacije in pošiljanje daljinskih ukazov na poljubno oddaljeno lokacijo.

▽

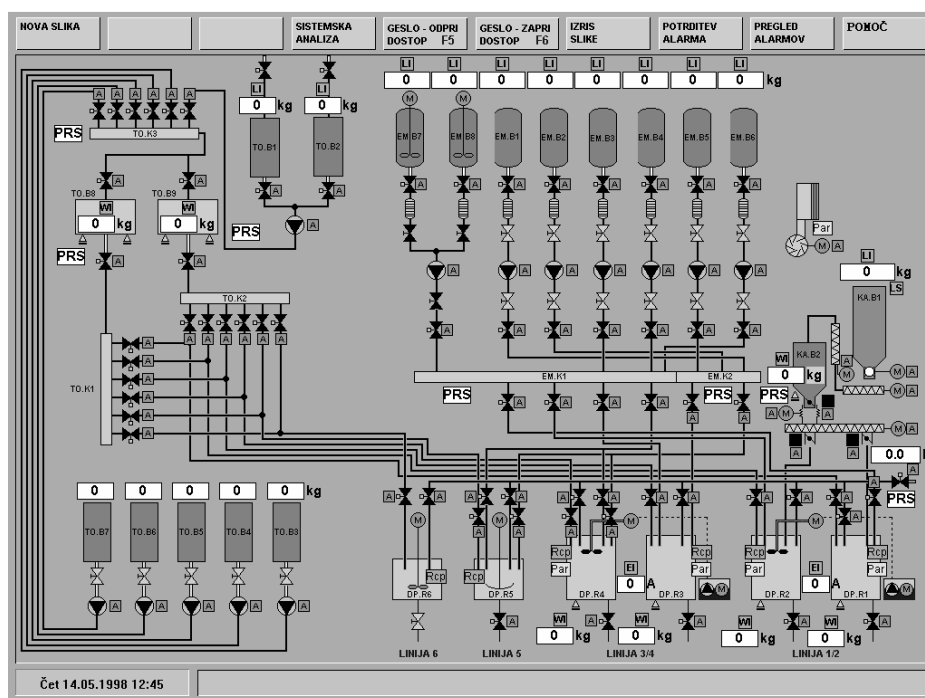
Primer 8.11: Uporaba orodja Factory Link

Kot primer je prikazan osnovni prikaz nadzornega sistema za vodenje proizvodnje polivinil acetatskih (PVA) lepil razvitega z orodjem Factory Link (Sl. 8.34).

Prikaz je definiran v grafični obliki s pomočjo objektov na osnovi tehnološke sheme procesa. Objekti predstavljajo posamezne procesne elemente, npr. ventil, črpalka, cisterna, itd.

Operater lahko preko prikaza spremlja stanje procesa s pomočjo naslednjih funkcij:

- številčni prikaz zveznih meritev;
- prikaz stanja ventilov, črpalk, motorja;
- s pomočjo barvanja poti se prikazuje vsebina medija;
- barvanje cistern glede na vsebino;
- položaj mešala v disperzijskih posodah DP1, DP2, DP3 in DP4.



Sl. 8.34. Grafično definiran prikaz na osnovi tehnološke sheme in pripadajočih objektov

Prav tako lahko operater spreminja način delovanja (ročno/avtomatsko) in s tem neposredno vpliva na proces. V spodnji vrstici je razvidno tudi obvestilo operaterju o pojavu alarma v procesu. Z aktivnimi gumbi v prvi vrstici pa izvede prehod na prikaz aplikacije, ki je razviden iz besedila na gumbu.

Stanje posameznega objekta je prikazano s pomočjo animacije objekta. Na prikazu *Tehnološka shema* (Sl. 8.34) se za posamezen ventil prikazuje stanje ventila na način prikazan v Tabeli 8.2.

Orodje Factory Link omogoča tudi upravljanje z recepti. Na Sl. 8.35 lahko vidimo ustrezen prikaz, ki je bil izdelan za delo z recepti v okviru že omenjenega sistema za vodenje proizvodnje PVA lepila.

Prikaz receptov omogoča operaterju izvrševanje naslednjih funkcij:

- vpis (kreiranje) recepta;
- shranjevanje recepta na disk;
- branje in brisanje recepta z diska;
- vpisa parametrov recepta v PLC;
- izvajanje operacij nad receptom (start, zadrži, nadaljui, stop, prekini, itd.).

Tabela 8.2. Prikaz stanja ventila na tehnološki shemi

Objekt	Barvno kodiranje		Stanje
VENTIL	Barvanje ventila	Črn	Zaprto
		Zelena	Odperto
		Rdeča	Napaka
		Rumen	Zapiranje ventila
		Moder	Odpiranje ventila

DP2 DP3 DP4 DP5 DP6 MF Zam.Cist. **DP1** Teh.schema Vpis Rcp Izris slike Pomoč

RECEPT: Stanje: Skupaj voda [l]: 0.0 Teža ob startu [kg]: 0

Naziv: Odpri PLC Start Zadrži Nadaljuj Stop

OPERACIJE: Trenutni čas: Čet 14.05.1998 13:38

Doziranje emulzije VPIS

Količina [kg]	P
Cist. Na razp. Zaht.	R
7 1350 1600	E
0 0 0	N
0 0 0	O
0 0 0	S
0 0 0	P
0 0 0	L
0 0 0	C

Zadrži Nadaljuj Stop Prekini Ločeno

PRIKAZ Stanje: Obratovanje

Količina [kg]	P
Cist. Na razp. Zaht. Doz.	T
7 1350 1600 0	R
0 0 0 0	O
0 0 0 0	D
0 0 0 0	I
0 0 0 0	T
0 0 0 0	E
0 0 0 0	V

RECEPTURA

Izpust topil VPIS

Kol.vode	Predviden	P
[l]	izpust	L
0 0		C

Zadrži Nadaljuj Stop Prekini Ločeno

PRIKAZ Stanje: Zaganjanje

Teht. Kol.topil	Kol.vode	Predviden	P
[kg]	[l]	izpust	T
0 0	0 0		R

Izpust kalcita VPIS

Kol.vode	Predviden	P
[l]	izpust	L
0 0		C

Zadrži Nadaljuj Stop Prekini Ločeno

PRIKAZ Stanje: Zaganjanje

Kol.kalcita	Kol.vode	Predviden	P
[kg]	[l]	izpust	T
0 0	0 0		R

Doziranje vode VPIS

Količina vode	P
[l]	L
0	C

Zadrži Nadaljuj Stop Prekini Ločeno

PRIKAZ Stanje: Začetno

Količina vode	P
[l]	T
0 0.0 0.0	R

Črpanje topil VPIS

Količina [kg]	P
Cist. Na razp. Zaht.	R
1 0 200	E
0 0 0	N
0 0 0	O
0 0 0	S
0 0 0	P
0 0 0	L
0 0 0	C

Zadrži Nadaljuj Stop Prekini Ločeno

PRIKAZ Stanje: Zaganjanje

Količina [kg]	P
Cist. Na razp. Zaht. Doz.	T
1 750 200 0	R
0 0 0 0	O
0 0 0 0	D
0 0 0 0	I
0 0 0 0	T
0 0 0 0	E
0 0 0 0	V

Črpanje kalcita VPIS

Količina [kg]	P
Na razp. Zaht.	L
550 800	C

Zadrži Nadaljuj Stop Prekini Ločeno

PRIKAZ Stanje: Zaganjanje

Količina [kg]	P
Na razp. Zaht. Doz.	T
550 800 0	R

Izpust iz disperzijske posode

PRIKAZ Stanje: Začetno

Količina v posodi	P
[kg]	T
0	R

Zadrži Nadaljuj Stop Prekini Ločeno

Sl. 8.35. Prikaz za upravljanje z recepti

Programska orodja, ki omogočajo razvoj in izvedbo aplikacij na nadzornih računalniških sistemih, t.i. orodja tipa SCADA so bila prvotno v funkciji podpore operaterju. Omogočala so razvoj aplikacij, katerih funkcije so predvsem zbiranje in arhiviranje procesnih podatkov, prikazi, alarmiranje, generiranje poročil, itd. V zadnjem obdobju se funkcije orodij SCADA širijo proti t.i. MES sistemom (angl. Manufacturing Execution Systems), ki podpirajo proizvodni nivo. Razvojni trendi so v nadgradnji tradicionalnih SCADA orodij, ki ohranjajo funkcije vmesnika med človekom in strojem.

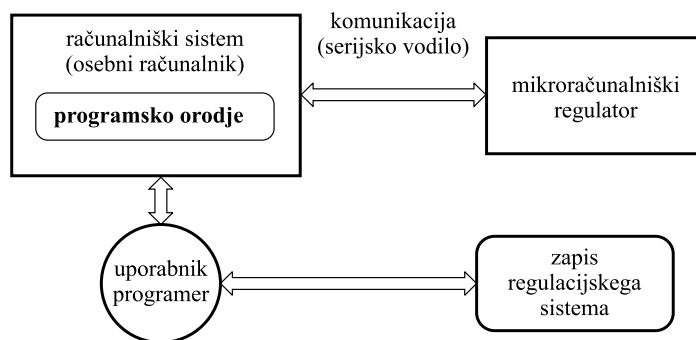
8.4.3 Programska orodja za regulatorje

Ta del poglavja obravnava programska orodja za konfiguriranje oziroma programiranje mikroračunalniških regulatorjev. Predstavljene so osnovne lastnosti programskih orodij in prikazane prednosti in koristi, ki jih prinaša njihova uporaba pri programiranju mikroračunalniških regulatorjev. Na koncu je podan tudi konkreten primer uporabe grafično orientiranega programskega orodja za programiranje mikroračunalniških regulatorjev.

8.4.3.1 Osnovne značilnosti programskih orodij za mikroračunalniške regulatorje

Mikroračunalniški regulatorji imajo, v nasprotju s starejšimi analognimi elektronskimi in pnevmatskimi regulatorji, zelo širok nabor funkcij ter različnih dodatnih operacij. Temu ustrezno je potrebno pred vključevanjem mikroračunalniškega regulatorja v proces izdelati oziroma prilagoditi delovni program, to je določiti ali izbrati konfiguracijo regulatorja in nastaviti vrsto različnih parametrov. Določitev konfiguracije in nastavitve parametrov se lahko izvede na dva različna načina. Prvi način temelji na uporabi tipk in prikazovalnikov čelne plošče regulatorja ali pa s posebno tipkovnico za programiranje. Programiranje poteka po principu izbiranja med vnaprej podanimi možnostmi in nastavljanja vrednosti določenih parametrov. Za izvedbo celotnega postopka programiranja navadno zadošča nekaj tipk in prikazovalnik z dvema vrsticama. Omenjeni način programiranja je sicer cenen in preprost, saj za izvedbo ne potrebujemo nobenega dodatnega orodja, vendar je delo nepregledno in utrudljivo, s tem pa se poveča možnost napak.

Alternativa omenjenemu načinu programiranja je programiranje s pomočjo ustreznih programskih orodij, ki so danes dostopna za praktično vsak tip mikroračunalniškega regulatorja. Pod pojmom *programsko orodje za mikroračunalniški regulator* razumemo programski paket, ki omogoča razvijanje oziroma izdelavo delovnega programa za določen mikroračunalniški regulator. Izhodišče postopka izdelave delovnega programa je običajno že obstoječ zapis regulacijskega sistema v določeni obliki, najprimerneje v obliki *bločnega diagrama*. Programsko orodje za regulator torej podpira preslikavo določene oblike zapisa regulacijskega sistema v ustrezen delovni program mikroračunalniškega regulatorja, kot ponazarja Sl. 8.36.



Sl. 8.36. Vloga programskega orodja

Mikroračunalniški regulatorji se medsebojno ločijo med drugim tudi po kompleksnosti programske opreme. S tem v zvezi obstajata dve skrajni možnosti:

- regulator je lahko *tovarniško pred-programiran* in ima običajno tovarniško vstavljen fiksni delovni program ali nekaj različnih delovnih programov, prilagojenih za reševanje tipičnih regulacijskih problemov. Uporabnik ima na delovanje tovarniško vstavljenih programov le minimalen vpliv, v glavnem lahko le izbira, kateri od tovarniško vstavljenih programov naj se izvaja, poleg tega pa lahko spreminja številčne parametre tega programa;
- regulator je lahko *prosto programirljiv* in je opremljen z uporabniku odprto programsko opremo, ki omogoča izdelavo poljubnih delovnih programov. Le-ti so do podrobnosti prilagojeni konkretnemu problemu vodenja, ki ga uporabnik rešuje. V tem primeru uporabnik-programer sam izdelava ustrezni delovni program. Pri izdelavi delovnega programa povezuje tovarniško pred-programirane funkcijske bloke oziroma podprograme, ki so sestavni del programske opreme regulatorja.

V primeru tovarniško pred-programiranih regulatorjev je postopek programiranja enostaven in se reducira samo na nastavljanje številskih parametrov in t.i. programskih stikal, to je parametrov, katerih zaloga vrednosti vsebuje le nekaj vrednosti. Celoten postopek je enostavno izvedljiv direktno na regulatorju s pomočjo tipk in prikazovalnikov, lahko pa uporabimo ustrezno programsko orodje. Programsko orodje je v tem primeru močno poenostavljeno in je podobno orodjem za obvladovanje preglednic ter deluje nad bolj ali manj fiksno bazo podatkov.

V primeru prosto programirljivih regulatorjev postopek izdelave uporabniških programov največkrat zahteva uporabo ustreznega programskega orodja. Tudi če je postopek izdelave delovnega programa mogoče izvesti direktno na regulatorju s pomočjo tipk in prikazovalnikov, predstavlja to težji in manj pregleden način v primerjavi z uporabo ustreznega programskega orodja. Programsko orodje je v tem primeru kompleksen programski paket, ki poleg ustrezne preglednice parametrov vsebuje še grafični urejevalnik, namenjen izdelavi (risanju) regulacijske sheme delovnega programa. S pomočjo grafičnega orodja uporabnik izdelava regulacijsko shemo,

pri tem pa uporablja pred-programirane funkcijske bloke iz knjižnice, ki jo vsebuje programsko orodje.

Pomembno je tudi poudariti, da uporabnik praviloma nima možnosti izbire med različnimi programskimi orodji, saj je le-to določeno že z vrsto in tipom izbranega mikroračunalniškega regulatorja. To pomeni, da široki ponudbi mikroračunalniških regulatorjev na trgu danes ustreza tudi širok izbor ustreznih programskih orodij, ki jih ponujajo posamezni proizvajalci mikroračunalniških regulatorjev. Primer programskega orodja je npr. paket SIPROM DR-24, ki ga ponuja Siemens za svoj prosto programirljivi regulator SIPART DR-24. Fischer&Porter za svoje regulatorje ponuja orodje 53 HC 2600 (Loop Master Configuration Software). Podobno Hartmann&Braun za regulatorje serije PROTRONIC nudi programsko orodje IBIS, itd.

8.4.3.2 Pregled opravil, ki jih podpirajo programska orodja

Različna programska orodja se glede na opravilno sposobnost medsebojno precej razlikujejo. Opravilna sposobnost je tako prilagojena programski opremi regulatorja, za katerega je namenjeno programsko orodje. Tipična opravila, ki jih podpirajo programska orodja, lahko strnemo v sledeče skupine:

- razvoj in izdelava novih delovnih programov;
- popravljanje ali dopolnjevanje obstoječih delovnih programov;
- prenos delovnih programov v regulator;
- shranjevanje delovnih programov;
- izdelava dokumentacije delovnih programov;
- tiskanje dokumentacije s pomočjo ustreznega tiskalnika.

8.4.3.3 Način uporabe programskih orodij

Sodobna orodja za prosto programirljive regulatorje so grafično orientirana. To pomeni, da uporabnik-programer delovni program dejansko nariše v obliki blokovne sheme, ki je podobna zapisu regulacijskih sistemov v obliki *bločnega diagrama*. Osnova za izdelavo delovnega programa regulatorja je torej bločni diagram načrtanega regulacijskega sistema. Programsko orodje pri tem omogoča enostaven način preslikave načrtanega bločnega diagrama v delovni program. Celoten postopek izdelave delovnega programa lahko strnemo v sledeče korake:

1. Izdelava blokovne sheme delovnega programa

Blokovna shema delovnega programa se izdelava s pomočjo grafičnega vmesnika, ki ga vsebuje programsko orodje. Pri izdelavi blokovne sheme uporabnik-programer izbira med tovarniško pred-programiranimi bloki iz knjižnice. Le-ti so v splošnem namenjeni izvajanju določenih funkcij, katerih pregled in opis bosta podana v nadaljevanju. Izbiri ustreznih funkcijskih blokov sledi njihovo prostorsko razmeščanje in medsebojno povezovanje. Vse opisane akcije uporabnik opravi z

interaktivno uporabo zaslona, miške in deloma tudi tipkovnice. Podrobnosti so seveda odvisne od vrste uporabljenega programskega orodja.

2. Vpisovanje vrednosti parametrov posameznih funkcijskih blokov

Izdelavi blokovne sheme delovnega programa sledi določanje vrednosti parametrov, ki pripadajo posameznim funkcijskim blokom in tako dokončno definirajo njihovo funkcijo in način delovanja. Kompleksnejšim funkcijskim blokom lahko pripada več parametrov. Nasprotno pa enostavni funkcijski bloki pogosto nimajo parametrov.

3. Določanje načina delovanja aparaturne opreme regulatorja

Uporabnik-programer v tej fazi določi:

- merilne obsege analognih vhodov in izhodov;
- način delovanja analognih in digitalnih prikazovalnikov;
- način delovanja regulatorja po koncu izpada napajalne napetosti;
- druge posebnosti v zvezi z delovanjem regulatorja.

4. Prenos delovnega programa v regulator

Ko je postopek izdelave delovnega programa končan, sledi prenos delovnega programa v regulator. Najpogostejši način povezave računalniškega sistema in regulatorja je t.i. serijski način komunikacije. V ta namen mora biti računalniški sistem opremljen z dodatnim serijskim vmesnikom. Nekatera programska orodja omogočajo tudi vzporedno priključitev večih regulatorjev na razvojni računalniški sistem. Pri prenosu delovnega programa v določen regulator je zato urejen ustrezen način naslavljanja posameznih regulatorjev.

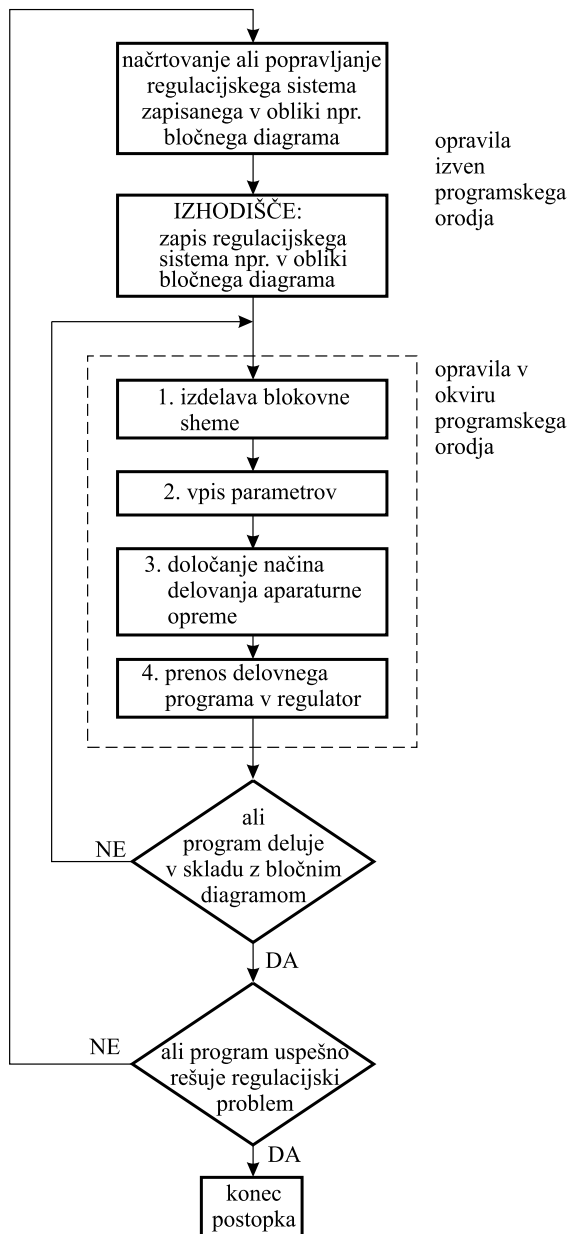
5. Dokumentiranje delovnega programa

Dokumentiranje delovnega programa se najpogosteje izvede avtomatsko. Dokumentacija delovnega programa vsebuje:

- grafični prikaz regulacijskega sistema v obliki blokovne sheme;
- spisek parametrov in njihovih vrednosti;
- grafični prikaz pomena tipk in prikazovalnikov čelne plošče;
- ustrezni okviri in glave shem.

Izdelava delovnega programa v splošnem ni premočrtni postopek, saj pridemo do delujočega delovnega programa navadno šele po določenem številu iteracij, ki vključujejo v tej knjigi že večkrat omenjeno *preverjanje* (verifikacijo) in *vrednotenje* (validacijo).

V fazi preverjanja ugotovimo, ali izdelani delovni program deluje v skladu s specifikacijami, ki jih podaja bločni diagram regulacijskega sistema. Če rezultat ni zadovoljiv, se je potrebno vrniti na izhodišče, kot prikazuje Sl. 8.37.



Sl. 8.37. Iterativni postopek izdelave delovnega programa

Če preverjanje ne izkazuje napak, sledi ovrednotenje delovnega programa, kjer ugotovimo, ali delovni program v skladu s postavljenimi zahtevami rešuje regulacijski primer. Če se pojavi odstopanje od zahtev, je potrebno ponovno pregledati postopek načrtovanja regulacijskega sistema. To pomeni, da je potrebno spremeniti izhodišče za izdelavo delovnega programa, t.j. zapis sistema v obliki bločnega diagrama.

8.4.3.4 Pregled tipičnih funkcijskih blokov

Nabor funkcijskih blokov je odvisen od vrste in tipa programskega orodja in regulatorja. Pomembno je, da nabor funkcijskih blokov omogoča enostavno in hitro preslikavo med zapisom regulacijskega sistema v delovni program regulatorja. V Tabeli 8.3 je podan tipičen nabor, ki ga najdemo pri večini programskih orodij. Funkcijski bloki so, glede na vrsto funkcije, ki jo izvajajo, razdeljeni v več razredov. V posameznih razredih se nahajajo t.i. tipični predstavniki funkcij, ki jih najdemo pri različnih tipih regulatorjev.

Tabela 8.3. Pregled tipičnih funkcijskih blokov

Razred	Funkcijski blok	Razred	Funkcijski blok
Aritmetične funkcije	Seštevanje signalov Odštevanje signalov Množenje signalov Deljenje signalov Potenciranje signalov Koren signalov Absolutna vrednost signala Interpolacija	Regulacijske funkcije	PID - blok, različne verzije Blok za generiranje časovnih potekov signalov
Logične funkcije	Logični IN Logični ALI Logični NeIN Logični NeALI Spominske celice Števec	Posebne funkcije	Številčni parametri Analogni prikazovalniki Digitalni prikazovalniki LE diode Tipke
Primerjalne funkcije	Primerjalnik signalov Omejilnik signalov Diferencialni ojačevalnik Maksimalna vrednost Minimalna vrednost Preklopnik signalov	Vhodi, izhodi	Analogni vhodi Analogni izhodi Digitalni vhodi Digitalni izhodi Komunikacijsko vodilo
Časovne funkcije	Integrator Diferenciator Filter Mrtvi čas Ura - timer		

Primer 8.12: *Uporaba programskega orodja SIPROM DR-24*

Podan je preprost primer uporabe grafično orientiranega programskega orodja Siemens SIPROM DR-24, ki je namenjeno razvoju delovnih programov za prosto programirljivi regulator Siemens SIPART DR-24. Prikazani program regulatorja se navezuje na primer regulacije temperature v toplotnem izmenjevalniku, ki je opisan v podpoglavju o PID regulaciji (Sl. 7.87). Način povezave regulatorja s procesom ter bločni diagram regulacijskega sistema sta podana v okviru primera v poglavju 9 na Sl. 9.37.

Regulacijski sistem meri izhodno temperaturo in pretok ogrevanega medija. Upoštevajoč odstopanje med želeno in izmerjeno vrednostjo temperature in upoštevajoč tudi velikost pretoka ogrevanega medija regulacijski sistem sproti določa potreben pretok pare, ki služi za ogrevanje. Referenčna vrednost temperature se lahko nastavi lokalno na regulatorju ali pa posredno preko analognega vhodnega signala. Omogočen je tudi preklon med avtomatskim in ročnim načinom delovanja regulatorja. Na Sl. 8.38 se nahaja originalna grafična dokumentacija programa za regulacijo temperature v toplotnem izmenjevalniku, ki je izdelan s pomočjo orodja SIPROM DR-24.

Na strani 1/1 je realiziran izbor med zunanjo in lokalno nastavljivo referenčno vrednostjo temperature. S pomočjo tipke *tA1*, ki je priključena na preklopnik signalov *ASo* se izbira med signalom zunanje referenčne vrednosti iz analognega vhoda *AE1* in med signalom lokalne referenčne vrednosti iz binarnega integratorja *bin1*. Kadar je aktiven signal lokalne referenčne vrednosti se vklopi dioda *L01*. Signal lokalne referenčne vrednosti iz binarnega integratorja *bin1* se povečuje oziroma zmanjšuje inkrementalno s pomočjo tipk *tA2* in *tA3*. Dejanska vrednost temperature se zajame s pomočjo analognega vhoda *AE2*. Referenčna in dejanska vrednost temperature se prikazujeta na digitalnih (alfa-numeričnih) prikazovalnikih *dd1* oziroma *dd2*. S pomočjo odštevalnika *Sub* se od referenčne vrednosti odšteje signal dejanske temperature, rezultat je signal regulacijske napake. Signal napake se vodi na stran 2/1 na regulacijski blok *Ccn1* in sicer na proporcionalni *XdP* in integralni *XdI* vhod (PI regulator). Nadalje je na vhod za direktno krmiljenje *Yz* priključen še linearno transformiran signal pretoka ogrevanega medija iz analognega vhoda *AE2*, ki predstavlja krmiljenje (feed-forward). Na blok *Ccn1* so priključene tudi tri tipke. S tipko *tA4* se izvaja preklon med ročnim in avtomatskim načinom delovanja regulacijskega bloka. Ročni način delovanja se označi z vklopom diode *L08*. Tipki *tA6* in *tA7* služita za inkrementalno povečevanje oziroma zmanjševanje izhoda regulacijskega bloka, kadar se le ta nahaja v ročnem režimu delovanja. Izhod regulacijskega bloka *Ccn1* je priključen na analogni izhod regulatorja *AA2* in pa na digitalni (alfa-numerični) prikazovalnik *dd3*.

- sodobna programska orodja so uporabniško prijazna, postopek izdelave programa je zato manj utrudljiv kot programiranje direktno na regulatorju;
- postopek izdelave delovnega programa s pomočjo programskega orodja je zaradi grafične orientiranosti zelo pregleden, kar pa ni slučaj pri direktnem programiranju na samem regulatorju;
- zaradi preglednosti je bistveno zmanjšana možnost napak;
- izredno se poenostavi naknadno popravljanje in dopolnjevanje delovnih programov;
- dokumentacija o delovnem programu se izvede avtomatsko.

Slaba stran orodij za regulatorje se odraža predvsem v njihovi razmeroma visoki nabavni ceni. Tako se nakup orodja v celoti izplača šele pri izdelavi delovnih programov za več regulatorjev.

Programiranje direktno na regulatorju brez uporabe programskega orodja je smiselno predvsem takrat, kadar se izvajajo manjši popravki na že obstoječem delovnem programu regulatorja, ki je že vključen v delovanje in je zato priključitev računalniškega sistema s programskim orodjem iz kakršnegakoli razloga otežena.

8.4.4 Programska orodja za krmilnike

Programabilni logični krmilniki (PLK) so bili razviti kot nadomestek relejne logike v poznih šestdesetih letih. Zato tradicionalni programski jeziki krmilnikov vsebujejo logične gradnike relejnih sistemov: relejne tuljave in kontakte, časovnike, števec in različne nabore logičnih Boolovih povezav. To omogoča enostavno in fleksibilno realizacijo sekvenčnega vodenja na PLK-jih.

Japonski in evropski proizvajalci krmilnikov so razvijali predvsem tekstovne programske jezike krmilnikov, oziroma tako imenovane instrukijske liste. Ameriški proizvajalci so želeli programiranje relejne logike čim bolj poenostaviti z ekvivalentnim grafičnim jezikom – lestvičnim diagramom. Zaradi svoje enostavnosti je lestvični diagram danes najbolj pogosto uporabljeni programski jezik za programiranje sekvenčnih algoritmov širom po svetu (J. R. Pollard, 1994).

Krmilniki so dolgo let predstavljali zaprt gradnik v sistemu avtomatizacije. Razlog je bil v različnosti krmilnikov pri njihovem načrtovanju in uvajanju, programiranju, dokumentiranju, medsebojnem komunikacijskem povezovanju in vzdrževanju. To je proizvajalce in uporabnike prisililo v standardizacijo omenjenega področja. Predvsem v Evropi je postavljeni standard dobil široko podporo širše strokovne javnosti. Bistvene točke standarda so opisane v nadaljnjem tekstu.

Čeprav so bili krmilniki rezultat razvoja nadomestka relejne logike, se z razvojem in cenitvijo mikroprocesorske opreme pojavljajo tudi na področju vodenja zveznih regulacijskih sistemov. Z uporabo ustreznih programskih orodij zamenjujejo množico eno- in več-zančnih regulatorjev.

8.4.4.1 Standard IEC 1131

Standardizacija PLK-jev je pomemben korak k odprti arhitekturi sistemov za avtomatizacijo. Mednarodna elektrotehniška organizacija IEC je v ta namen leta 1983 ustanovila podgrupo z namenom, da se pripravi široko zasnovan standard za krmilnike. Leta 1987 je bil javnosti podan prvi predlog standarda, leta 1993 pa je komisija standard potrdila kot *IEC 1131*.

Namen standarda IEC 1131 je poenotenje: dela, uporabe, programiranja in dokumentiranja pri uporabi različnih vrst PLK-jev. Uporaba standarda vodi v zmanjševanje porabe človeških virov na področju učenja, načrtovanja, vzdrževanja in svetovanja pri uporabi krmilnikov. Omogoča, da se uporabnik posveti problemu vodenja, in ima možnost ponovne uporabe že znanih uspešnih rešitev. Standard zmanjšuje možnosti napak in nerazumevanja pri prenosu znanja in tehnologije vodenja s krmilniki.

Elementi standarda

Standard vsebuje pet osnovnih elementov:

- *IEC 1131-1* opisuje splošni del, kjer so definirani izrazi, ki nastopajo v standardu, tako za strojno kot za programsko opremo PLK-jev;
- *IEC 1131-2* definira strojno opremo PLK-jev in njegovih enot – modulov;
- *IEC 1131-3* opisuje programiranje PLK-jev: programske jezike in skupne elemente;
- *IEC 1131-4* definira zahtevano uporabniško dokumentacijo, tako za strojni kot za programski del;
- *IEC 1131-5* definira komunikacijo PLK-jev med seboj in z drugimi standardnimi enotami za avtomatizacijo. Komunikacija je podrobneje opisana v samostojnih standardih (Profibus, Ethernet, MAP, itd).

Tretji del, IEC 1131-3, je bil deležen posebne pozornosti načrtovalcev standarda. Definira pet programskih jezikov, štirim že uveljavljenim PLK-jevim jezikom so dodali še petega, za višje nivojsko programiranje.

Programski viri standarda

Tretji del, to je programski del standarda IEC 1131-3, definira nekaj pomembnih elementov, ki jih bomo podali v nadaljevanju (PLCopen, 1995).

Sintaksa in semantika petih jezikov

Standard predvideva *hkratno uporabo* različnih programskih jezikov na istem krmilniku povezanih v en izvršni program. Standard ne zahteva, da programska orodja za programiranje PLK-jev podpirajo vseh pet programskih jezikov. Sintaksa in semantika jezikov je točno definirana in ne dopušča dialektov.

Podatkovni tipi in spremenljivke

Podatkovni tipi preprečujejo vnos napak v program že na začetni stopnji. Uporabljajo se za definicijo vsakega parametra in spremenljivke v programu. Spremenljivke so normalno omejene na posamezno programsko organizacijsko enoto, v kateri so definirane, torej so lokalne. Če morajo imeti spremenljivke globalen doseg, jih moramo tako deklarirati (VAR_GLOBAL). Parametrom se lahko določi vrednost, ki jo bodo imeli pri startu ali hladnem zagonu krmilnika.

Konfiguracija, viri in opravila

Programska struktura je razdeljena na tri nivoje. Na najvišjem nivoju, to je reševanje zadanega problema vodenja, nastopa konfiguracija. Konfiguracija zajema posebnosti krmilnika na strojnem področju (procesor, pomnilnik, V/I naslovni prostor, itd). Konfiguracija je na vsakem krmilniku ena sama in vsebuje enega ali več virov. Viri vsebujejo opravila, ki preko upravljalca opravil izvršuje kodo IEC programa periodično ali na predpisan dogodek.

Programske organizacijske enote POU

POU (Program Organization Units) so samostojni programski moduli, ki sestavljajo opravila. Končni program je zbirka POU-ov razporejenih v različnih opravilih. Vsak POU je lahko napisan v svojem IEC programskem jeziku.

Programski jeziki standarda

Razvijalci standarda so se ozirli na že znane in uveljavljene rešitve, zato so vključili v standard štiri najpogostejše uporabljane programske jezike in mu dodali še peti visoko nivojski jezik. Programski jeziki standarda IEC 1131-3 so naslednji:

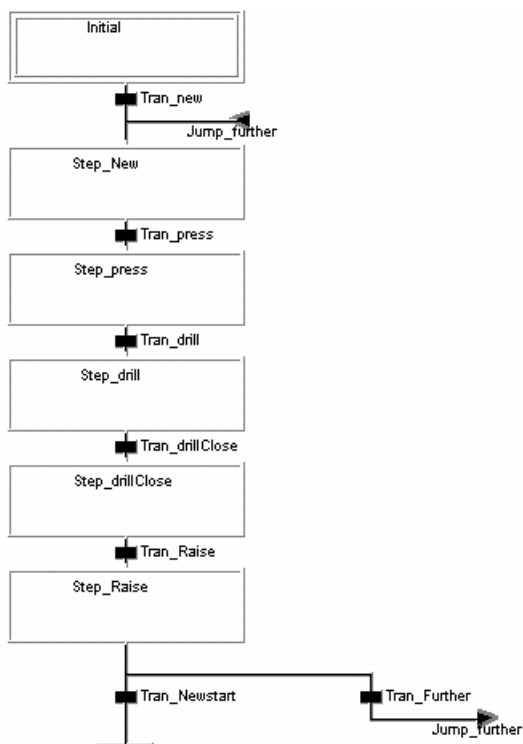
SFC - sekvenčna funkcijska shema (Sequential Function Chart)

Razvil se je iz znanega Grafcet-a (IEC 848), ki je v bistvu diagram poteka, in je pregleden tudi pri kompleksnih programskih strukturah. Programski tok je definiran s *koraki* in *prehodi*. Izvajanje sekvenčnega koraka je odvisno od pogoja prehoda. Če je pogoj prehoda izpolnjen, se nadaljuje z izvrševanjem naslednjega koraka, pri neizpolnjenem pogoju, pa se programski tok na tistem mestu zaustavi, do izpolnitve pogoja. SFC tvori osnovo med jeziki standarda IEC 1131-3, vendar ne more nastopati samostojno. Koraki in prehodi so napisani v ostalih programskih jezikih, npr. v lestvičnem diagramu. Zaradi svoje enostavnosti nudi dobro stično točko ljudem različnih usmeritev in znanj (elektronika, procesno vodenje, robotika, vzdrževanje).

▽

Primer 8.13: Sekvenčna funkcijska shema

Enostavna SFC shema za vodenje obdelovalnega stroja je za ilustracijo prikazana na Sl. 8.39. SFC uporablja grafični programski vmesnik.



Sl. 8.39. Primer sekvenčne funkcijske sheme (SFC) s koraki in prehodi.

Δ

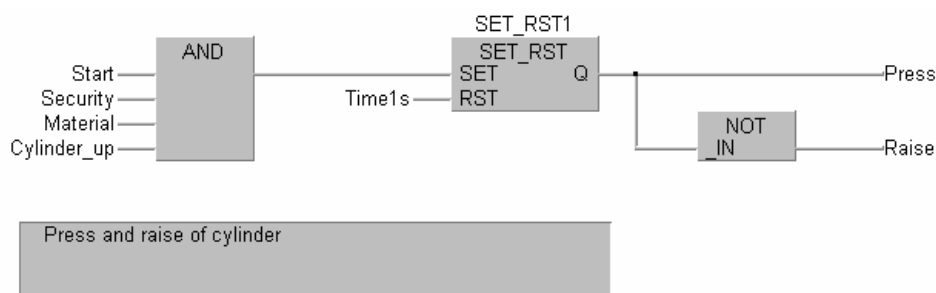
FBD – funkcijski blokovni diagram (Function Block Diagram)

Elementi jezika so *bloki*, ki jih povezujemo med seboj kot npr. električno shemo. Omogoča uporabo pripravljenih blokov iz programskih knjižnic iz različnih področij (procesno vodenje, vodenje pozicije orodij, komunikacije, itd). Omogoča hkratno uporabo in povezavo z lestvičnim diagramom. Uporablja grafični programski vmesnik.

▽

Primer 8.14: Funkcijski blokovni diagram

Primer vodenja cilindra stiskalnice s funkcijskim blokovnim diagramom je prikazan na Sl. 8.40.



Sl. 8.40. Primer funkcijskega blokovnega diagrama (FBD)

Δ

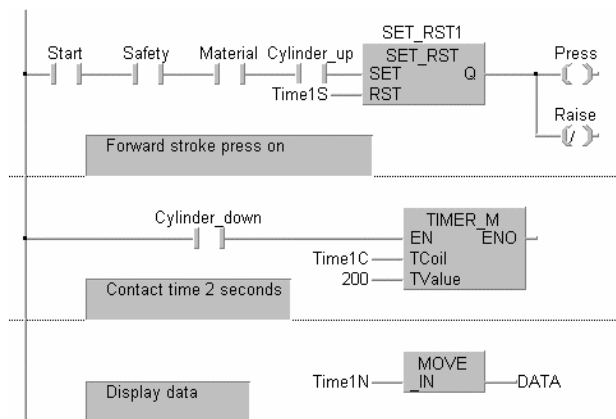
LD – lestvični diagram (Ladder Diagram)

Lestvični diagram je najbolj razširjeno orodje za programiranje sekvenčnih funkcij na PLK-jih. Zasnovan je na grafični predstavitvi relejnih logičnih shem. Uporablja grafični programski vmesnik. Standard IEC 1131-3 dopušča hkratno uporabo funkcijskih blokov in relejnih simbolov v lestvičnem diagramu, kot je to razvidno iz primera na Sl. 8.41.

▽

Primer 8.15: Lestvični diagram

Na Sl. 8.41 je prikazan primer lestvičnega diagrama. Prva vrsta lestvičnega diagrama prikazuje enakovredno vodenje cilindra stiskalnice kot FBD na Sl. 8.40.



Sl. 8.41. Primer lestvičnega diagrama (LD)

Δ

IL – instrukcijska lista (Instruction List)

Instrukcijska lista je programski jezik podoben zbirniku. Je nepregleden, zato je primeren predvsem za manjše aplikacije ali optimizacijo dela programa.



Primer 8.16: *Instrukcijska lista*

Na Sl. 8.42 je prikazan primer instrukcijske liste. Z instrukcijsko listo je realiziran primer vodenja cilindra stiskalnice, ki smo ga realizirali že z uporabo funkcijskega blokovnega diagrama in lestvičnega diagrama.

LD	Start
AND	Security
AND	Material
AND	Cylinder_up
ST	Result
CAL	SET_RST1(EN:=TRUE,SET:=Result,RST:=Time1S,Q:=Press)
LDN	Press
ST	Raise

LD	Cylinder_down
Timer_M	Time1C, 200

Sl. 8.42. Primer instrukcijske liste (IL).



ST – strukturiran tekst (Structured Text)

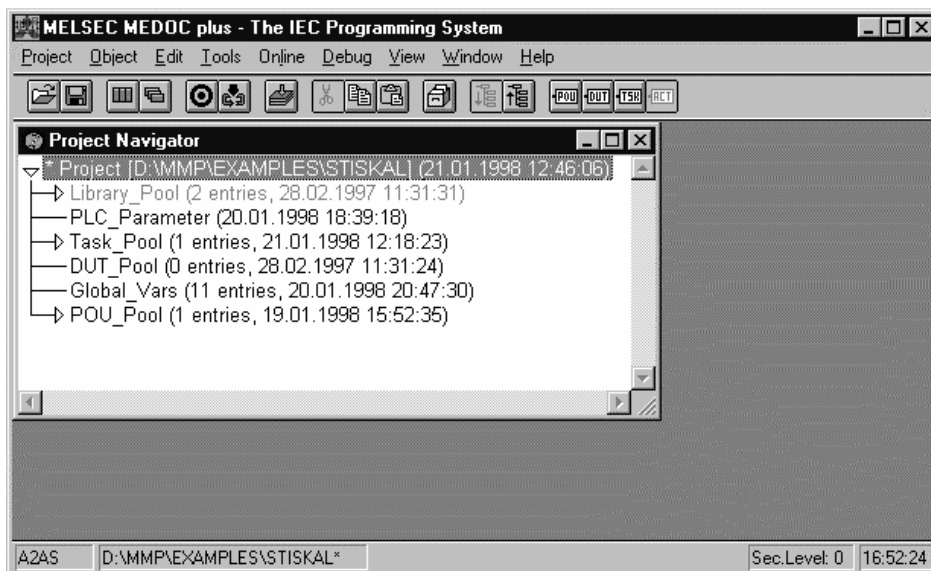
Strukturiran tekst je nov jezik razvit posebej za standard IEC 1131-3. To je visokonivojski jezik z sintakso podobno Pascal-u. Uporablja se za kompleksne procedure, ki jih ne moremo realizirati z grafičnimi jeziki. Uporablja se tekstovni vmesnik.

8.4.4.2 Programsko orodje MELSEC MEDOC plus

Danes je na tržišču že več programskih paketov, ki omogočajo programiranje PLK-jev po standardu IEC. Če naštejemo le nekatere: MELSEC MEDOC plus, Step 7, RSLogix, SYSWIN, PARADYM-31, ISaGRAF in drugi. V nadaljevanju bo opisano delo z programskim paketom MELSEC MEDOC plus (v nadaljevanju MM+), ki je namenjen programiranju krmilnikov Mitsubishi. Njegovo delovanje bo opisano na primeru vodenja stiskalnice iz poglavja 7.3.2 Sekvenčno vodenje.



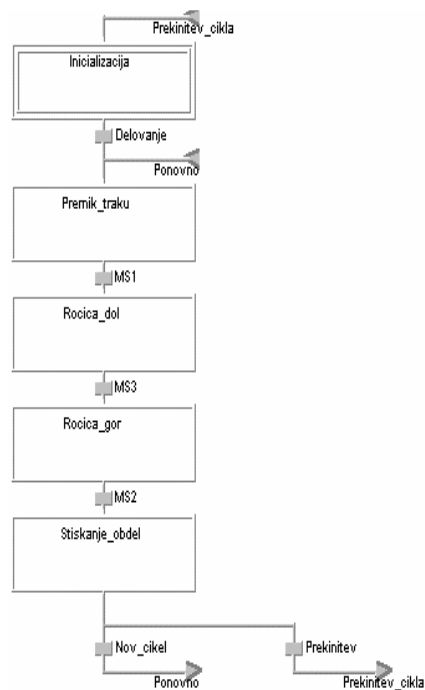
Primer 8.17: *Izdelava programa za vodenje stiskalnice s programskim paketom MM+*



Sl. 8.43. MM+ in okno projektnega navigatorja z strukturo projekta

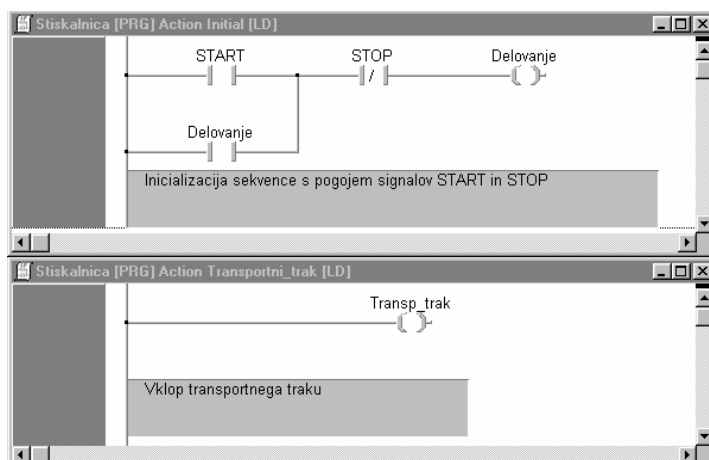
Pri programiranju se držimo naslednjih korakov:

- odpiranje novega projekta STISKAL in izbira tipa krmilnika (npr. A2AS). Na zaslonu se prikaže okno *Project Navigator* (Sl. 8.43) s standardno drevesno strukturo projekta: ime projekta, programska knjižnica standarda in proizvajalca krmilnika, konfiguracija krmilnika, področje opravil, področje podatkovnih tipov, globalne spremenljivke in področje programskih organizacijskih enot;
- deklariranje globalnih spremenljivk projekta. Vsaki globalni spremenljivki se določi enolični identifikator (npr. *Transp_trak*), naslov (npr. *%QX16*), tip (npr. *BOOL*), začetno vrednost (npr. *FALSE*) in komentar (npr. *Pogon transportnega traku*);
- programiranje POU vedno vsebuje zaglavje in telo. Pri odpiranju novega POU določimo ali bo to program, funkcija ali funkcijski blok. V primeru stiskalnice smo izbrali program. Nato izberemo programski jezik za POU (npr. *Sequential Function Chart*).
- deklariranje zaglavja (header) POU. Tu so definirane globalne in lokalne spremenljivke, ki jih uporabimo v telesu POU;
- programiranje telesa POU, kjer se nahaja dejanski program krmilnika. V primeru stiskalnice smo izbrali za jezik SFC, ki je zaradi svoje preglednosti najbolj primeren za sekvenčna krmilja (Sl. 8.44). Vsakemu koraku v SFC programu določimo akcijo, ki se izvrši po predpisani sekvenci in izpolnjevanje pogojev prehoda med koraki. Prehodi med koraki so lahko kar logična stanja spremenljivk oziroma je to določena programska struktura;



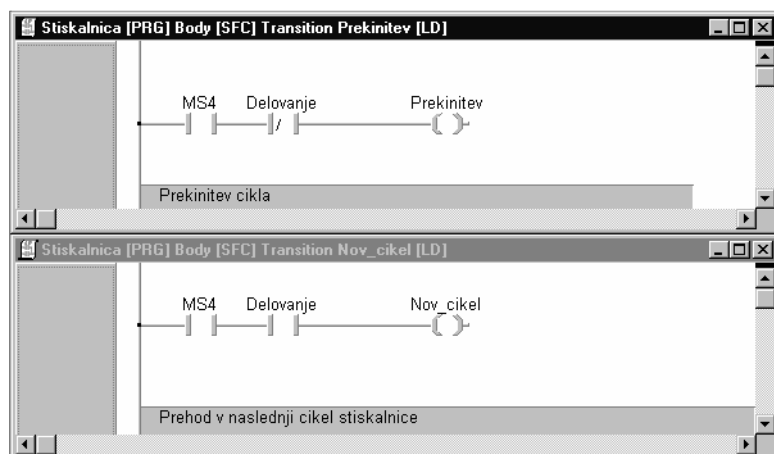
Sl. 8.44. Telo programske organizacijske enote POU - program stiskalnice napisan v SFC jeziku

- programiranje akcije koraka se izvede v jeziku IL, LD ali FBD. Za primer stiskalnice je bil uporabljen jezik LD. Za dva koraka 'Inicijalizacija' in 'Premik_traku' je prikazan primer na Sl. 8.45. Programi akcije se znotraj istega POU priredijo posameznemu koraku SFC programa;



Sl. 8.45. Programiranje korakov iz SFC -ja 'Inicijalizacija' in 'Premik_traku' v lestvičnem diagramu

- prehodi med koraki so samo logična stanja “pravilno” in “nepravilno”. Stanja prehoda so lahko kar digitalni vhodi (npr. mejno stikalo na pnevmatski ročici stiskalnice MS3) ali pa programi napisani v IL, LD ali FBD jeziku. Rezultat programa prehoda mora biti logično stanje z eno izhodno spremenljivko, ki je ime prehoda. Primer programov dveh prehodov napisanih v LD za stiskalnico prikazuje Sl. 8.46;



Sl. 8.46. Program za prehod 'Prekinitev' in 'Nov_cikel'

- razvrščanje POU med opravila. Različne POU lahko razvrstimo v posamezno opravilo, ki se izvršujejo na tri različne načine: na dogodek, ciklično na določen interval ali s prekinitvijo. Določi se tudi prioriteta opravila, če se hkrati pojavi več zahtev po izvrševanju različnih opravil;
- prevajanje programa: preveri se sintaksa, generira se izvršna koda za krmilnik. Če program vsebuje sintaktične napake, se izpiše lista napak in njihovo mesto v programu;
- prenos programa v PLK in njegovo testiranje. MM+ omogoča sprotni (online) nadzor izvrševanja in tudi sprotno popravljanje programa.

Δ

8.4.4.3 Blokovni regulacijski jezik IDR BLOK

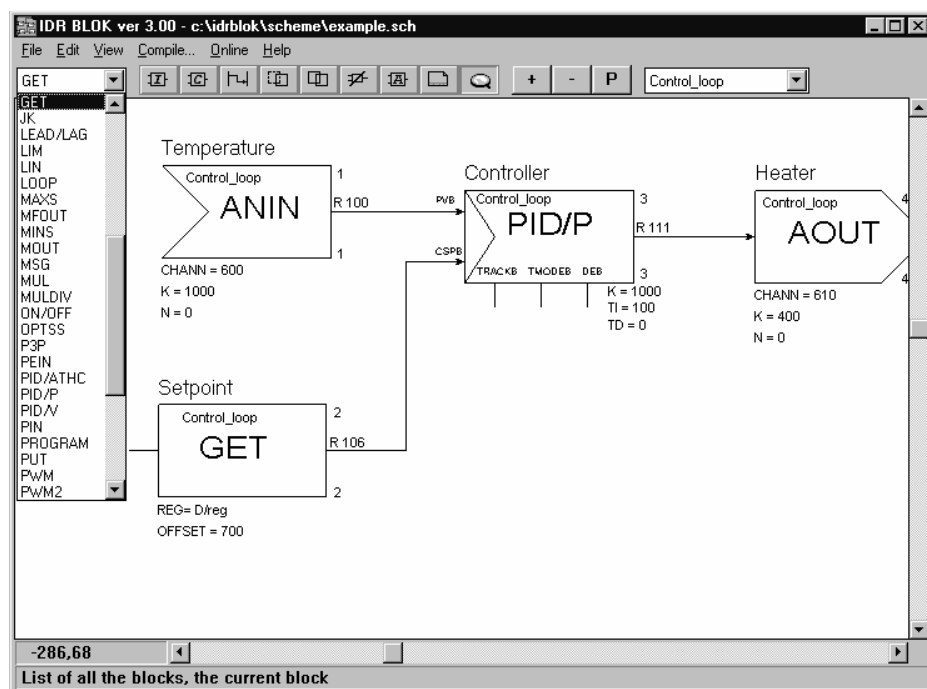
V predhodnem poglavju je bil opisan standard IEC 1131-3, ki je prvenstveno namenjen za sekvenčno in logično krmiljenje naprav diskretnega tipa. V praksi se izkaže, da so industrijski sistemi delno diskretni in delno zveznega značaja. Čeprav omogoča uporaba funkcijskih blokovnih diagramov tudi zvezno regulacijo, se izkaže, da so specializirana orodja za vodenje zveznih sistemov bolj uspešna. Razlog je v hitrejšem načrtovanju programov z bolj specializiranimi bloki, potrebnimi za različne primere regulacij ter kvalitetnejših podpornih funkcijah: 'on-line' spremljanje in spreminjanje programa,

časovni diagrami in optimizacija programske kode za manj zmogljive PLK-je.

Eden izmed takšnih specializiranih jezikov je tudi IDR BLOK proizvajalca INEA, Domžale, v sodelovanju Instituta "Jožef Stefan" in Fakultete za elektrotehniko v Ljubljani. To je jezik za vodenje zveznih procesov s krmilniki Mitsubishi. Osnova je programski jezik, ki je problemsko orientiran za regulacije in vodenje različnih zveznih procesov. Uporablja se ga z enostavnim grafičnim vmesnikom, s katerim sestavljamo regulacijsko blokovno shemo. Uporaba IDR BLOK-a omogoča direktno digitalno regulacijo DDC (Direct Digital Control) na sorazmerno cenenih in zanesljivih krmilnikih, ki so bogato podprti z vhodno/izhodnimi moduli in različnimi vrstami standardnih komunikacij (Profibus, Ethernet, MAP, RS232, RS485, itd).

Izdelava programa temelji na povezovanju različnih blokov in dodeljevanju parametrov (atributov) tem blokom. Tak način načrtovanja programa, je zaradi blokovne strukture najbližji inženirju avtomatiku, saj si logično sledijo bloki s funkcijami: meritve, preoblikovanje signalov, regulacije in izhodi na izvršilne člene. Na poti se pojavijo še pomožne funkcije, ki so lahko povezane s sekvenčnim vodenjem (zagon/ zaustavitev naprave, alarmna stanja, varnostne funkcije, itd) ali pa z ukazi operaterja na nadzornem sistemu (Marinšek, 1996).

Primer razvoja aplikativnega programa v IDR BLOK-u je prikazan na Sl. 8.47.



Sl. 8.47. Izbira in vnos bloka iz knjižnice blokov.

Lastnosti programskega jezika IDR BLOK

Aplikativni program napisan v IDR BLOK-u se sestoji iz grupe blokov (LOOP), ki tvorijo programsko zanko z enako periodo izvajanja. Blok je osnovni element programa.

Tipi blokov so razdeljeni v šest skupin:

- *vhodni bloki* opravljajo zajem signalnih vrednosti iz procesa in njihovo transformacijo v inženirske enote;
- *funkcijski bloki* predstavljajo različne odločitvene funkcije, npr. iskanje minimalne ali maksimalne vrednosti, izbiranje ene vrednosti izmed štirih, itd.;
- *aritmetični in logični bloki* izvršujejo aritmetične in logične operacije;
- *regulacijski bloki* vsebujejo različne PID regulatorje, dvo- ali tro-točkovne regulatorje, odprtozančni krmilni blok, itd.;
- *izhodni bloki* pošiljajo izhodne vrednosti na aktuatorje;
- *pomožni bloki* izvršujejo različne pomožne operacije, kot so omejevanje vrednosti, izmenjevanje podatkov s sekvenčnim programom, itd.

Lastnosti vgrajenih regulacijskih algoritmov

Klasična povratnozančna regulacija s PID regulatorjem je v praksi najpogosteje uporabljana regulacija. Vgrajena sta dva osnovna tipa PID regulatorjev:

- *PID/P* je pozicijski regulator za vodenje izvršnih členov z absolutno pozicijo. Uporablja se pri pnevmatskih ventilih, motornih ventilih s pozicionerjem ali pri aktuatorjih s pulzno-širinsko moduliranim signalom (PWM).
- *PID/V* je hitrostni regulator z inkrementalnim delovanjem. Uporablja se za vodenje izvršnih členov z inkrementalno pozicijo. Taki so npr. motorni ventili brez meritve pozicije, z diskretnim signalom za odpiranje in zapiranje ventila.

Lastnosti vgrajenih PID regulatorjev močno vplivajo na kvaliteto regulacije v povratnozančnih sistemih. Najpomembnejše lastnosti so:

- brezudarni preklap iz avtomatskega v ročni režim. To omogoča enostavno realizacijo tudi ročno vodenega sistema, npr. iz oddaljenega nadzornega sistema;
- vgrajen algoritem za preprečevanje integralnega pobega regulatorja. Ta lastnost je pomembna prevsem v sistemih, ki niso optimalno načrtovani, in se izvršni člen pogosto nahaja v skrajnem položaju (popolnoma odprt ali popolnoma zaprt);
- prosto določljivi vhod za diferencialni del regulatorja. Diferencialni del (D) je lahko povezan na regulacijski pogrešek, na regulirano vrednost ali pa na nek drug signal, ki je npr. povezan z meritvijo motnje v regulacijski zanki;
- z različnimi vezavami PID regulatorjev se lahko izvedejo kompleksne regulacijske strukture, kot so npr. kaskadne regulacije ali regulacije razmerja.

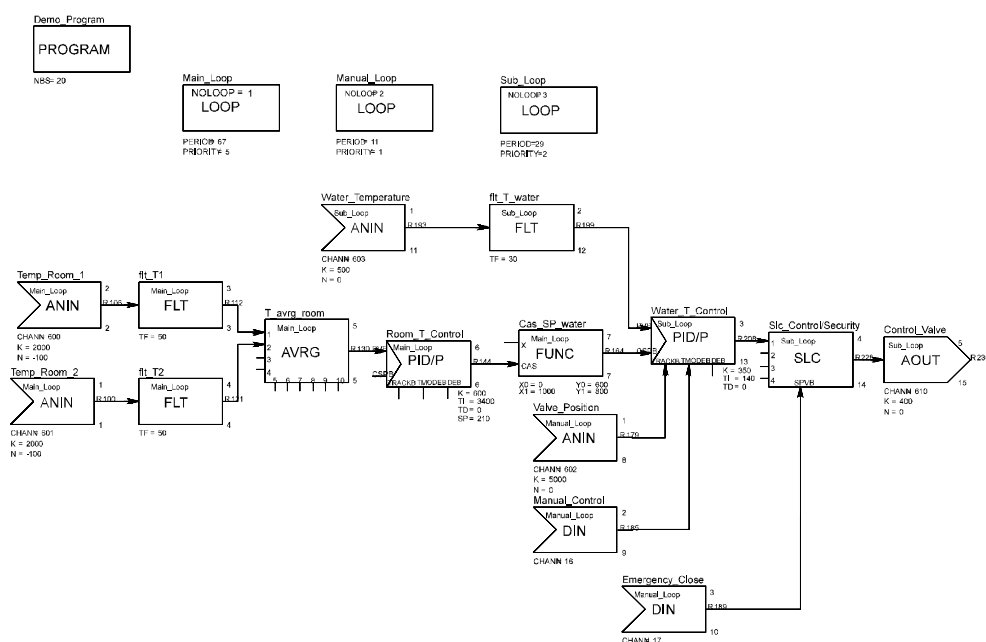
Poleg klasičnih PID regulatorjev je mogoče uporabiti tudi krmilne zanke (feed forward)

in regulatorje entalpije, za optimalno vodenje klimatskih naprav. Novejši regulacijski bloki vključujejo samonastavljivi PID regulator in mehki (fuzzy) regulator. Mehki regulator ima svoj uporabniški vmesnik in editor za konfiguracijo ter preizkušanje.

▽

Primer 8.18: Izvedba kaskadne regulacije temperature z jezikom IDR BLOK

Potrebno je načrtati regulacijski algoritem za vodenje temperature v prostoru. Temperatura v prostoru se regulira z vročevodnim ogrevalnim sistemom s temperaturo vode od 60 do 80°C z mešalnim regulacijskim ventilom v kotlovnici. Podana mora biti možnost ročnega vodenja regulacijskega ventila in avtomatskega zapiranja v primeru alarma (pregrevanje sistema). Imamo dve enakovredni meritvi temperature prostora, temperaturo ogrevne vode in vrednost dajalnika ročne pozicije ventila. Dva kontrolna diskretna signala postavljata zahtevo za ročno vodenje ventila in za alarmno zapiranje ventila. Izhodni krmilni signal je zvezen s pozicijsko vrednostjo ventila.



Sl. 8.48. Primer regulacijske sheme v IDR BLOK-u.

Rešitev problema z uporabo kaskadne regulacije prikazuje Sl. 8.48. Vrednosti temperatur prostora se filtrirajo z digitalnim filtrom (FLT), nato se izračuna aritmetična vrednost obeh meritev (AVRG). Rezultat služi kot procesna veličina za prvi regulator, ki regulira temperaturo prostora. Prvemu regulatorju je kaskadno podrejen regulator temperature vode, ki mu preko linearne funkcije (FUNC) spreminja referenčno temperaturo vode od 60 do 80°C. Z izbirni blokom (SLC) se zapira ventil ob alarmu. Ročno vodenje ventila je realizirano na regulatorju temperature ogrevalne vode.

8.5 Zaključek

V poglavju smo podali in opisali nekaj tipičnih predstavnikov različnih skupin orodij, ki jih srečujemo pri razvoju in izgradnji sistemov za vodenje. Zavedati se je potrebno, da je različnih orodij, ki so dosegljiva na trgu zelo veliko, in da je večina od njih zelo podvržena spremembam, ki jih narekuje hiter razvoj računalniške tehnologije. Zato je pri nakupu potrebna premišljena politika, ki mora zagotavljati kontinuiteto in stabilnost dela.

Prav na koncu pa omenimo še problem, ki ga dosedaj nismo načeli. Kot smo videli, se pri nekaterih orodjih, posebej v skupini, ki se nanaša na konfiguriranje in implementacijo funkcij vodenja, pojavlja dilema ali gre za orodje ali za programske gradnike. Najbolj je ta dilema mogoče izražena pri SCADA programskih paketih in ekspertnih lupinah. Ugotovimo lahko, da v naših primerih zelo jasne ločnice med orodjem in gradnikom pravzaprav ni, oziroma da določen gradnik kar vsebuje orodje za svojo konfiguracijo. Pogosta je tudi situacija, ko imamo dve različici istega programskega paketa. Prva je "razvojna verzija", ki ima bolj funkcijo orodja, druga pa "izvršna" (run-time) verzija, ki je bolj podobna gradniku. To pomeni, da bi v tej knjigi, nekatera orodja, ki smo jih opisali v tem poglavju prav lahko uvrstili v naslednje poglavje, ki se nanaša na gradnike. V teh primerih je torej delitev na orodja in gradnike nekoliko umetna, vendar pa smo jo zaradi konsistentnosti knjige vseeno obdržali.

