REGULA - An Interactive User Oriented Language for Implementation of Control Systems

Magnusson, Boris; Löfgren, Mats; Elmqvist, Hilding; Fernström, Christer; Kruzela, Ivan; Schönthal, Tomas

1981

Document Version:
Förlagets slutgiltiga version

CODEN:LUTFD2/(TFRT-7213)/1-027/(1981)

REGULA - An Interactive User Oriented Language for Implementation of Control Systems.

Magnusson, B
Löfgren, M
Elmqvist, H
Fernström, C
Kruzela, I
Schönthal, T

LUND INSTITUTE OF TECHNOLOGY
DEPARTMENT OF AUTOMATIC CONTROL
FEBRUARY, 1981

Organization
LUND INSTITUTE OF TECHNOLOGY
Dept of Automatic Control
Box 725
S-220 07 LUND 7
Sweden

Document name

Date of issue
February, 1981

CODEN: LUTFD2/(TFRT-7213)/1-027/(1981)

Author(s)
Magnusson, B, Löfgren, M, Elmqvist, H,
Fernström, C, Kruzela, I, Schönthal, T

Sponsoring organization

Title and subtitle

REGULA - An Interactive User Oriented Language
for Implementation of Control Systems.

A4 A5

Abstract

When designing and testing an automatic control system it is extremely valuable to be able to modify the system interactively while the process is running. Regula has been designed to meet this demand for interaction.

Regulators consist of a set of interconnected nodes. A node type is defined in a declaration part and an algorithm part. The declaration part contains the definition of input, output and state variables. The algorithm part is defined with assignment statements, if desired including conditional expressions.

Several nodes can be created of each type. Connections between nodes and to the external process are made in a way similar to the block diagrams often used in this field.

While the process is controlled it is possible to declare new node types, to change the algorithms of a node, to create new nodes and to change the connections between nodes.

An implementation of Regula has been made for a micro computer, LSI-11, with 28 k 16 bit words of memory. The programming languages used for the implementation are Concurrent Pascal and Sequential Pascal.

The report is written in Swedish.

Key words A4 A5

Interactive, Direct Digital Control, Concurrent Pascal,
Application Language.

Classification system and/or index terms (if any)

Supplementary bibliographical information

Language
Swedish

ISSN and key title

ISBN

Recipient's notes

Number of pages
27

Price

Security classification

Distribution by (name and address)

REGULA - An Interactive User Oriented Language for Implementation of Control Systems.

Abstract:

When designing and testing an automatic control system it is extremely valuable to be able to modify the system interactively while the process is running. Regula has been designed to meet this demand for interaction.

Regulators consist of a set of interconnected nodes. A node type is defined in a declaration part and an algorithm part. The declaration part contains the definition of input, output and state variables. The algorithm part is defined with assignment statements, if desired including conditional expressions.

Several nodes can be created of each type. Connections between nodes and to the external process are made in a way similar to the block diagrams often used in this field.

While the process is controlled it is possible to declare new node types, to change the algorithms of a node, to create new nodes and to change the connections between nodes.

An implementation of Regula has been made for a micro computer, LSI-11, with 28 k 16 bit words of memory. The programming languages used for the implementation are Concurrent Pascal and Sequential Pascal.

The report is written in Swedish.

REGULA - Ett interaktivt applikationsspråk för implementation av reglersystem.

Innehållsförteckning:

Projektledare:

Hilding Elmqvist

Deltagare:

Christer Fernström
Ivan Kruzela
Mats Löfgren
Boris Magnusson
Tomas Schönthal

1. Introduktion

Reglersystem på dator implementeras ofta med hjälp av så kallade DDC-paket (Direct Digital Control). Beskrivningen av reglersystemet organiseras då vanligen som ett antal fristående delar, noder, där varje nod har en avgränsad uppgift t ex digital filtrering eller PID-regulering. Noderna samverkar genom att utsignaler från en nod kopplas som insignaler till en annnan nod. Knytningen till den externa processen görs via A/D- och D/A-omvandlare eller binära in- och utgångar. Algoritmerna i noderna genomlöps med ett visst intervall, eventuellt varierande mellan olika noder.

Vid konstruktion och uttestning av ett reglersystem är det värdefullt att under pågående reglering kunna förändra reglersystemet med avseende på antal noder, sammankoppling av noderna och algoritmen i de olika noderna. Härigenom undvikes det återkommande, ofta besvärliga och tidsödande, uppstartningsförloppet.

Det är vidare önskvärt att sammankopplingen av noderna kan formuleras på ett sätt som överensstämmer så väl som möjligt med de blockscheman reglerteknikern ofta använder. I ett blockschema beskrivs reglersystemet uppdelat i flera samverkande noder. Uppdelning gör att även stora system går att överblicka och hantera. Beskrivningen av noderna delas upp i två nivåer, dels beskrivningen av nodernas yttre karaktär och dels deras inre funktion. Väldefinierade tvärsnitt mellan noderna gör att det på ett kontrollerat sätt går att ändra deras inre funktion. Detta sätt att arbeta har i hög grad bestämt utformningen av Regula där nodernas tvärsnitt utåt respektive inre beteende definierats var för sig.

Traditionellt skrivs DDC-paket i t ex Fortran. Varje typ av reglernod motsvaras då av ett underprogram. En förändring av algoritmen i en nod medför att motsvarande underprogram måste kompileras om och DDC-paketet länkas om. En sådan operation innebär vanligen att det reglerade systemet måste

stoppas och därefter åter startas upp med det nya reglersystemet.

Vid utformningen av Regula har stor vikt lagts vid att möjliggöra en högre grad av interaktivitet än vanligt. I Regula går det att ändra algoritmerna i noderna under pågående reglering, så väl som att skapa helt nya nodtyper, generera nya noder och ändra sammankopplingen.

2. Beskrivning av Regula

Notationen för beskrivning av reglernoder i Regula påminner om notationen i Algol. Beskrivningen av en nodtyp är uppdelad i en statisk deklarationsdel och en föränderlig algoritmdel. Syntaxen för språket finns i ett appendix.

Deklarationsdelen innehåller deklarationer av in- och utsignaler, tillståndsvariabler samt parametrar. Detta utgör en definition av nodens gränsyta mot övriga noder.

In- och utsignaler används för kommunikation med andra noder och med omvärlden. Parametrar används för att två noder av samma nodtyp ska kunna få något olika beteende. Parametrarna är skyddade så att de ej kan förändras av algoritmen, men kan däremot ändras av operatören. Tillståndsvariabler innehåller information som behövs vid nästa exekvering av noden. Alla typerna av variabler kan ges begynnelsevärden.

Exempel på en nodtyps-deklaration:

```
node type FILTER;
   input I;
   output U;
   state OLDI:=0;
end;
```

Algoritm-delen innehåller tilldelningssatser, eventuellt innehållande villkorliga uttryck. Det finns inga repetitiva satser. Detta är dock ingen större brist eftersom de flesta regleralgoritmer naturligt kan formuleras som en sekvens av tilldelningssatser. Lokala variabler och konstanter kan också utnyttjas i algoritmen. De vanliga standardfunktionerna för trigonometri, kvadrot och logaritm ingår i språket. Alla variabler är av reell typ.

Exempel på en algoritm (ett enkelt filter):

```
begin node FILTER;
   U:=(I+OLDI)/2;
   OLDI:=I;
end;
```

Den enkla språkstrukturen gör att Regula-systemet kan kontrollera att variabler fått värden innan de används. Det kan inte uppstå oändliga slingor eller andra programmeringsfel med katastrofala effekter. Språket är därmed helt säkert i den meningen att programmeringsfel inte kan spoliera pågående reglering. Däremot kan naturligtvis olämpliga algoritmer ställa till problem.

Man kan skapa en eller flera noder av en viss nodtyp. Dessa kan då ha samma eller olika intervall mellan exekveringarna. Sammankopplingen av noderna görs enligt exemplet nedan och beskriver det blockschema reglerteknikern ritar.

```
regulator EXAMPLE;
   node FILTER(5) F1;
   F1.I    <- anin.2;
   anout.1 <- F1.U;
end;
```

I exemplet skapas en nod av typen FILTER med exekverings intervallet 5. Tidsenheten är tiden mellan två klockavbrott, i vår implementation 20 ms. Sammankoppling har skett så att noden F1:s insignal I får sitt värde från analog ingång 2, och analog utgång 1 får sitt värde från noden F1:s utsignal U. Om flera noder skapas sker exekveringen av dem i den ordning de skapats i regulatorbeskrivningen. Om de dessutom har samma exekveringsintervall kommer denna ordning att bibehållas.

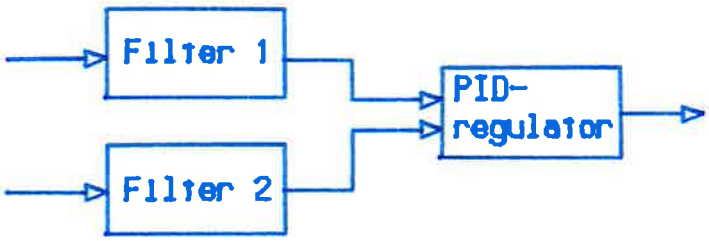
Under pågående reglering kan man skapa nya noder och ändra sammankopplingen. Man kan lägga in ett digitalt filter på en ingång, byta en reglernod till en mer komplicerad nod etc. Man kan också ändra exekveringsintervallet för en exister-

ande nod. Restriktionen att deklarationsdelen inte kan förändras betyder inte så mycket då det går att deklarera helt nya nodtyper. Noder av en ny typ kan helt enkelt ersätta noder av en äldre typ.

I Regula ingår förutom en reglertekniker-konsol, där beskrivningar och ändringar av reglersystemet utförs, även en processoperatör-konsol där värden på in- och utvariabler, tillstånd samt parametrar för en nod kan avläsas. Denna senare del har dock inte implementerats inom projektet.

3. Exempel på beskrivning av reglersystem

Följande reglersystem skall implementeras:



I Regula beskrives detta enligt följande:

Filtrens deklarations- och algoritm-delar:

```
node type FILTERTYPE;
   input IN;
   output OUT;
   state OLDIN:=0;
   param WEIGHT:=0.5;
end;

begin node FILTERTYPE;
   OUT   := WEIGHT*IN + (1-WEIGHT)*OLDIN;
   OLDIN:= IN;
end;
```

PID-regulatorns deklarations- och algoritm-delar:

```
node type PIDREGTYPE;
   input Y, REF;
   output U;
   state I:=0, YOLD:=0;
   param GAIN:=1, TI:=1E10, TD:=0, DT:=5;
end;
```

```
begin node PIDREGTYPE;
    var E;
    E:=YREF-Y;
    U:=GAIN*E + I - TD*(Y-YOLD)/DT;
    I:=I+E/TI*DT;
    YOLD:=Y;
end;
```

I följande regulatorbeskrivning skapas två noder av typen FILTERTYPE och en av typen PIDREGTYPE. Noderna kopplas samman enligt blockschemat och yttre signaler ansluts:

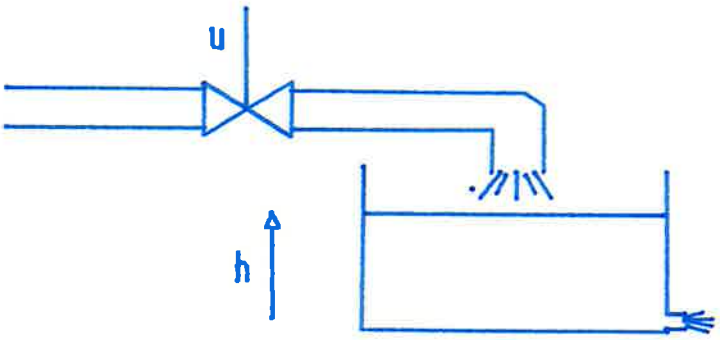
```
regulator EXSYST;
    node FILTERTYPE(5) FILTER1, FILTER2;
    node PIDREGTYPE(5) PID;

    FILTER1.IN <- anin.1;
    FILTER2.IN <- anin.2;
    PID.Y      <- FILTER1.OUT;
    PID.REF    <- FILTER2.OUT;
    anout.1    <- PID.U;
end;
```

Efter dessa operationer är regleringen startad. Det är nu möjligt att ändra algoritmerna för de olika nodtyperna och omedelbart avläsa vilken effekt detta får på regleringen av den externa processen. Om man inte är nöjd med en PID-regulator kan man beskriva en mer avancerad regulatortyp och länka in en sådan nod istället för PID-regulatorn. Som ett enklare exempel placerar vi in ett filter även på utgången. Detta görs genom att regulator beskrivningen utökas till följande:

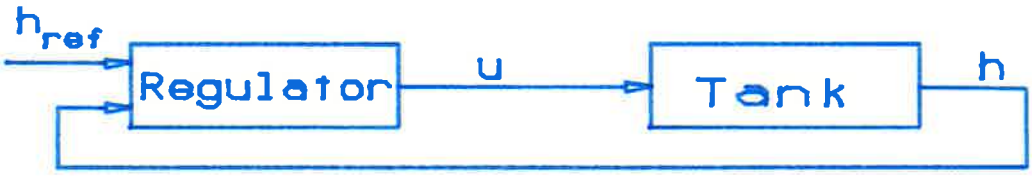
```
regulator EXSYST;
    node FILTERTYPE(5) FILTER1, FILTER2, OUTFILTER;
    node PIDREGTYPE(5) PID;

    FILTER1.IN    <- anin.1;
    FILTER2.IN    <- anin.2;
    PID.Y         <- FILTER1.OUT;
    PID.REF       <- FILTER2.OUT;
    OUTFILTER.IN <- PID.U;
    anout.1       <- OUTFILTER.OUT;
 end;
```

Vi skall nu ge ett exempel där flexibiliteten hos Regula-systemet är mycket värdefull för att process-ingenjören skall kunna införa sina kunskaper om processen i reglersystemet.

Nivån i en tank skall regleras med en inloppsventil.

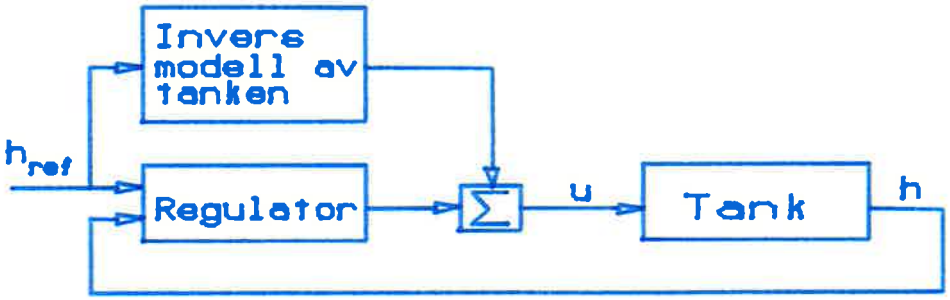


Denna reglering kan göras enligt följande blockdiagram.



Reguleringen kan t ex ske med en PID-regulator. Regleringen försvåras emellertid av att tanken har starkt olinjära egenskaper. Man skulle eventuellt behöva använda olika förstärkning i regulatorn vid olika arbetspunkter. Vidare måste PID-regulatorn behandla integraldelen i regleralgoritmen med hänsyn tagen till att styrsignalen är begränsad.

Ett alternativt sätt att införa regleringen framgår av nedanstående blockdiagram.



Meningen är att stora förändringar av h_{ref} direkt skall tas om hand av den inversa modellen (framkoppling). För att hantera störningar och ofullkomligheter i den inversa modellen används återkoppling via en enkel regulator.

En enkel dynamisk modell av tanken är

$$\frac{dh}{dt} = \frac{q_{in}}{A} - \frac{a}{A}\sqrt{2gh}$$

där a=utloppsarean och A=tankens bottenarea

Venitler har ofta kvadratisk karaktäristik varför inloppsflödet q_{in} ges av

$$q_{in} = k*(u-u_0)^2$$

Man kan för regleringen använda en invers stationär modell. Genom att sätta dh/dt=0 fås framkopplingstermen h_{ff}

$$q = a\sqrt{2gh_{ref}} \qquad u_{ff} = \sqrt{\frac{q}{k}} + u_0$$

En regulator med denna framkoppling samt en P-regulator för återkoppling kan beskrivas på följande sätt i Regula.

```
node type TANKREG;
   in  Y, YREF;
   out U;
end;

begin node TANKREG;
   const GAIN=2, G=9.81, a=9.5E-5, K=25E-6, U0=0.12;
   var   HREF, H,Q,UFF,V,E;

   HREF := (YREF-0.1)/0.4*0.45;
   H    := (Y   -0.1)/0.4*0.45;
   Q    := a*sqrt( 2*G* max(HREF,0) );
   UFF  := sqrt(Q/K)+U0;
   E    := HREF-H;
   V    := GAIN*E+UFF;
   U    := V*0.4 + 0.1;
end;
```

Beskrivningen har för enkelhets skull gjorts som endast en nodtyp. I algoritmen har även inkluderats omräkning av mätsignalerna till ingenjörsenheter.

Detta exempel visar fördelen med att processingenjören enkelt kan ange regleralgortimer eftersom hans kunskap om processen på det sättet kan programmeras in. Det är vidare viktigt att detta kan göras inkrementellt utan att regleringen av andra delprocesser störs, och att man på detta sätt enkelt kan prova olika lösningar tills en tillfredsställande reglering uppnås.

4. Regula vid terminalen

Beskrivningen av ett reglersystem i Regula består av tre olika typer av moduler:

node type - deklerationsdelen för en nodtyp
begin node - algoritm för en nodtyp
regulator - generering och sammankoppling av noder

För att enkelt kunna ändra i ett reglersystem görs beskrivningen i separata textavsnitt som hanteras var för sig. Dessa beskrivningar lagras som filer på sekundärminne och kan på kommando redigeras på bildskärm. För redigeringen utnyttjas en intelligent terminal med lokala redigeringsmöjligheter. Av det skälet får varje regulatorbeskrivning maximalt bestå av 22 rader. Detta är dock ingen allvarlig begränsning eftersom flera satser kan skrivas på samma rad, åtskilda av semikolon.

När en beskrivning har förberetts på skärmen begärs överföring till datorn för kompilering. Kompileringen styrs med funktionstangenter.

F1: kompilering av node type beskrivning
F2: kompilering av begin node beskrivning
F3: kompilering av regulator beskrivning

Om fel hittas vid kompilering lämnas felmeddelande och vidare bearbetning av beskrivningen avbryts. Först när inga fel påträffas vid kompilering infogas en ny beskrivning i reglersystemet.

En deklaration av en nodtyp måste ha kompilerats innan motsvarande algoritm komipleras eller nodtypen omnämnes i en regulator beskrivning. Deklarationer av nodtyper kan inte ändras, däremot kan deras algoritmdel och regulator beskrivningar ändras efter önskan.

Beskrivningar som inte för ögonblicket bearbetas av operatören lagras som filer på sekundärminne. Begäran om överföring av en regulatorbeskrivning från en fil till skärmen respektive från skärmen till en fil styrs med två funktionstangenter:

F9: (get file) fil överförs till skärmen
F10: (save file) skärminnehållet överförs till fil

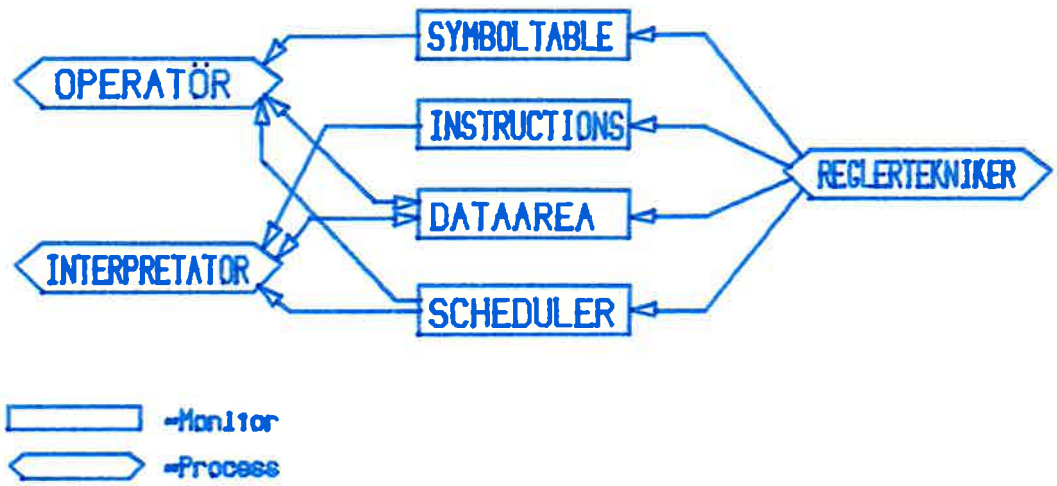
Filnamn anges på en kommandorad, märkt med en prompt '>'. Standardformat för filnamn används.

5. Implementation

Regula har implementerats på en i reglersammanhang mycket vanlig dator, LSI-11, en mikrodator med ett primärminne med 28 k 16-bits ord tillgängligt för operativsystem och användarprogram. Operativsystemet RT-11 utnyttjades.

Regula paketet har skrivits i Concurrent Pascal och sekventiell Pascal. Strukturen i Concurrent Pascal med monitorer och processer har naturligtvis påverkat systemlösningen. Färdiga biblioteksprogram har utnyttjats för in- och utmatning av analoga signaler.

Den dynamik som finns i Regula ställer stora krav på implementationen av systemet. Förändring av algoritmer under pågående reglering möjliggörs genom att algoritmerna interpreteras. Interpretatorn som exekverar algoritmerna i noderna, beskrivs som en fristående process. Kommunikationen med process-operatören och reglerteknikern ses också som fristående processer. Vi skiljer på fyra olika datatyper, implementerade som monitorer, genom vilka processerna kommunicerar. Nedan beskrivs först processerna och därefter monitorerna.



REGLERTEKNIKER

Reglerteknikerns aktiviteter kan indelas i fyra olika fall:

1) Deklaration av en ny nodtyp.

En ny nodtyp införs i SYMBOLTABLE med information om tillståndsvariabler, in- och utsignaler samt parametrar. Variablernas namn lagras och numreras. Detta är den grundläggande informationen som karaktäriserar en nodtyp. Variablernas startvärden lagras också här.

2) Specifikation av en algoritm.

Algoritmen skrivs i ett språk som influerats av Algol. Språket är emellertid starkt begränsat och endast tilldelningssatser finns. Algoritmen översätts till en assemblyliknande pseudokod lämplig för interpretering och lagras i monitorn INSTRUCTIONS. Översättningen sker i den omgivning som definierats genom deklarationen av nodtypen under punkt 1.

3) Generering av noder.

Det går att skapa flera upplagor, noder, av varje nodtyp. Dessa kommer då att genomlöpa samma algoritm, men med olika data. För varje nod reserveras ett data-utrymme i DATAAREA i enlighet med deklarationen under punkt 1. Uppgifter om skapade noder och hur ofta en nod skall exekveras införes i monitorn SCHEDULER.

4) Sammankoppling av genererade noder.

Sammankoppling av noder sker genom att man för en insignal i en nod anger varifrån dess värde skall hämtas. Det går också att koppla insignaler till A/D-omvandlare och utsignaler till D/A-omvandlare. Informationen om hur signalerna är kopplade lagras i DATAAREA tillsammans med nodernas data-utrymme.

INTERPRETATOR

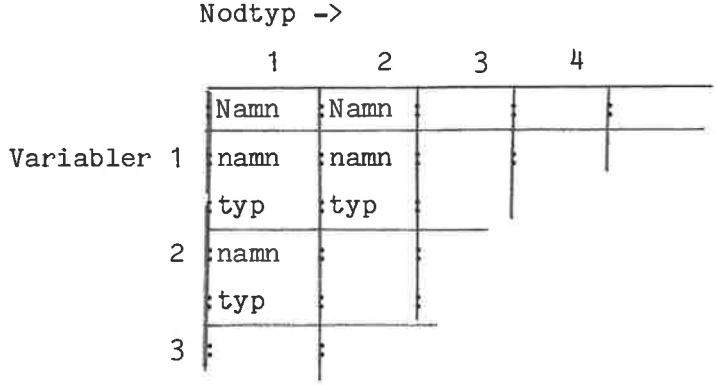
Interpretatorn arbetar cykliskt med att exekvera algoritmen för en nod i taget. Genom anrop av SCHEDULER erhålles besked om vilken nod som står näst i tur och när den skall exekveras. Vid denna tidpunkt hämtas den till typen hörande algoritmen från INSTRUCTIONS och nodens data-utrymme från DATAAREA varefter algoritmen genomlöps. Avslutningsvis återlämnas det uppdaterade data-utrymmet till DATAAREA.

OPERATÖR (ej implementerad)

Operatören anger med ett namn vilken nod han vill studera. Processen OPERATÖR presenterar då värdena på nodens variabler på en terminal. OPERATÖR anropar SCHEDULER för att få nodens nodtyp. Från SYMBOLTABLE kan därefter namn och typ på nodens variabler erhållas. Genom anrop av DATAAREA hämtas variablernas aktuella värden vilka presenteras på skärmen. Värdena uppdateras med lämpligt intervall. Operatören kan också ändra värden på nodens tillståndsvariabler och parametrar. På grund av tekniska problem att ansluta mer än en terminal till den använda datorn implementerades inte denna process inom projektet.

SYMBOLTABLE

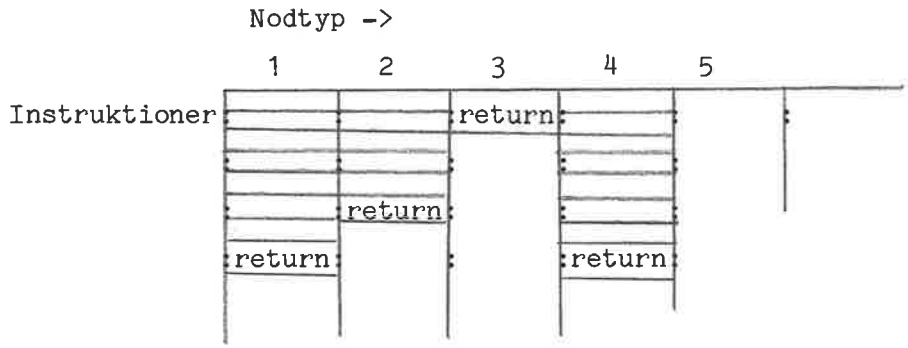
I SYMBOLTABLE finns för varje nodtyp namnet samt en förteckning över dess variabler med uppgift om typ, ordningsnummer och namn. Denna information är organiserad som ett fält av nodtyper där varje nodtyp innehåller namn på nodtypen och ett fält av variabler.



Nodtyperna fylls på i ordning och blir därigenom numrerade, likaså får varje variabel ett nummer inom nodtypen. Dessa nummer används av övriga delar av systemet för att identifiera nodtyper och variabler.

INSTRUCTIONS

Monitorn INSTRUCTIONS är organiserad som ett fält av kod. Kod i sin tur är ett fält av instruktioner.



Nodtyperna är numrerade i samma ordning som i SYMBOLTABLE. Instruktionerna refererar till variabler i en nods data-utrymme. De anges med nummer enligt numreringen av variablerna i SYMBOLTABLE.

Vid översättningen av en algoritm som inte innehåller några fel lagras den nya översättningen i fältet. Om en nodtyp inte innehåller någon algoritm har den i sin kod endast ett återhopp. Det språk som används för att representera algoritmerna redovisas i appendix.

DATAAREA

I DATAAREA finns två datastrukturer. Dels ett fält av data-utrymme och dels ett fält av länkar. I data-utrymmet reserveras plats för varje genererad nod. En numrering av

noderna används för fortsatt identifiering av de enskilda noderna. I fältet med länkar markeras för varje in-variabel varifrån dess värde skall hämtas. För utvariabler länkade till D/A-omvandlare lagras motsvarande information.

När värdena i ett data-utrymme hämtas från DATAAREA beräknas värdena av dess insignaler. Eventuella insignaler från A/D-omvandlare samplas. När sedan data-utrymmet återlämnas överförs värden från eventuella utsignaler, knutna till D/A-omvandlare, till respektive utgång.

SCHEDULER

I SCHEDULER representeras varje genererad nod av ett element i en länkad lista. Noderna är sorterade efter tid till nästa exekvering. Efter varje order om exekvering till INTERPRETATOR flyttas motsvarande nod bakåt i listan. För varje nod finns uppgifter om namn, nodtyp-nummer, data-area-nummer och exekverings-intervall. Listan är implementerad på traditionellt Fortran sätt eftersom det i Concurrent Pascal saknas både pekare och dynamisk minnesallokering.

På grund av den statiska minnesallokeringen i Concurrent Pascal har datastrukturerna ovan implementerats som fält. Detta gör att redan vid kompileringen av programpaketet måste man fixera gränser för antalet nodtyper, antalet variabler i en nodtyp, antalet noder, längden av koden för en algoritm etc. Det starkt begränsade minnesutrymmet gör att dessa fält inte kan göras stora nog för varje rimligt reglersystem. I stället måste programpaketet ibland justeras för att passa en speciell tillämpning som t ex innehåller lite längre algoritmer än vanligt, fler variabler etc.

6. Utvigningar av Regula

Regleralgoritmer kan ofta natuligt beskrivas med en sekvens av tilldelningssatser. I reglersystem ingår emellertid ofta förreglingsstyrning och sekvensstyrning, varvid binära mät- och styrsignaler används. Koppling till den externa proccessen behöver då också kunna ske med binära signaler.

Binär styrning görs förnärvarande ofta med så kallade PC-system. Dessa är enkla 1-bits datorer som repetitivt läser in binära mätsignaler, evaluerar en uppsättning av Booleska uttryck samt ställer ut binära styrsignaler.

I Regula kan förreglingsstyrning beskrivas i noder där algoritmen består av en uppsättning Booleska tilldelningssatser. Man behöver då också logiska variabler. I den befintliga implementationen av Regula finns bara variabler av reell typ, men vi kan representera true och false som 1.o respektive 0.o.

Sekvensstyrning kan beskrivas på samma sätt som i PC-system om man inför tillståndsvariabler med vars hjälp man håller reda på i vilket steg av sekvensen processen befinner sig. Det är dock naturligare att mera explicit ange att exekveringen av en nod skall fördröjas tills något villkor är uppfyllt. Detta kan uttryckas med:

```
wait until <Booleskt uttryck>
```

Villkoret testas vid varje samplingstidpunkt för noden.

Om villkoret är komplicerat behöver man kunna använda hjälpvariabler. Vidare är det lämpligt att med varje väntetillstånd associera ett namn som kan användas vid operatörskommunikation. Formen på väntesatsen blir då:

```
wait in <tillståndsnamn>
        tilldelningssatser
until <Booleskt uttryck>
```

Vid sekvensstyrning har man även behov av att kunna ange fördröjning en viss tid. Detta kan uttryckas med:

wait in <tillståndsnamn> for <tid>

Genom att införa de ovan beskrivna satserna kan man på ett enhetligt sätt beskriva kombinerade regler- och styrsystem.

7. Erfarenheter från projektet

Projektet utfördes som en del av en forskarkurs vid institutionen för Reglerteknik vid Lunds tekniska högskola. I projektet medverkade forskarstuderande från institutionerna för Elautomatik, Informationsbehandling och Reglerteknik.

Arbetet ufördes vid reglertekniks mikrodatorsystem av typ LSI-11. Erfarenheten av dessa system är mycket god då de visade sig kunna klara av även relativt besvärliga regleringar utan alltför stort overhead i exekveringstid. Systemet var också lätt att använda vid programutvecklingen. Tillgängligt primärminne var dock något knappt. Trots att programpaketet segmenterades kan restriktionen på storleken av de reglersystem som kan implementeras vara besvärande.

Användningen av ett högnivåspråk för implementationen var en förutsättning för att arbetet skulle kunna utföras inom ramen för ett projekt. Concurrent Pascal har använts för de realtids-beroende delarna och visat sig vara ett värdefullt redskap. Konstellationen med ett interpreterat Concurrent Pascal system och ett kompilerat Pascal system visade sig dock vara lite svårarbetad.

Utnyttjandet av Concurrent Pascal har också influerat uppläggningen av reglerspråket Regula självt. Det grundläggande begreppet nodtyp liknar mycket det vanliga typbegreppet i Pascal. Konstruktionen med en algoritmdel associerad till varje nodtyp ger dock analogi med processbegreppet i Concurrent Pascal. Synkroniserings-mekanismerna i Regula har emellertid kunnat göras väsentligt enklare. Detta beror på att algoritmerna i Regula begränsats till endast sekvenser av tilldelningssatser. Detta gör att man kan tillåta att noderna exekverar en och en från början till slut. Det väldefinierade sätt varigenom noderna tillåts kommunicera gör också att endast den till noden associerade data-arean behöver skyddas under det att algoritmen för noden genomlöps.

Appendix: Syntax för Regula

LEXICAL ITEMS

```
<ident> ::= <letter> {<letter> | <digit>}
<number> ::= <integer> [.<integer>] [ E [+|-] <integer>]
<integer> ::= {<digit>}
```

BASIC ITEMS

```
<variable> ::= <ident>
<constant> ::= <ident>
<node type ident> ::= <ident>
<node ident> ::= <ident>
<regulator ident> ::= <ident>
```

NODE TYPE

```
<node type> ::=
  node type <node type ident>;
  { input <variable list>; |
    output <variable list>; |
    state <variable list>; |
    param <variable list>; }
  end

<variable list> ::= <variable> [ := [+|-] <number> ]
                  { ,<variable> [ := [+|-] <number> ] }
```

NODE BODY

```
<node body> ::=
  begin node <node type ident>;
  { const <constant> = [+|-] <number>
      { , <constant> = [+|-] <number> } ;
    var <variable> { , <variable> } ; }

  { <assignment>; }

  end
```

ASSIGNMENT

```
<assignment> ::= <variable> := <expression>; |
                 <variable> = <Boolean expression>;
```

EXPRESSION

```
<expression> ::= <simple expression> |
                 if <Boolean expression> then
                 <simple expression> else <expression>
<simple expression> ::= [<add op>] <term> {<add op> <term>}
<add op> ::= + | -
<term> ::= <factor> {<mult op> <factor>}
<mult op> ::= * | /
<factor> ::= <number> | <variable> | <constant> |
             (<expression>) |
             <function id 1> (<expression> |
             <function id 2> (<expression>, <expression>)
<function id 1> ::= sin | cos | arctan | abs |
                    ln | exp | sqrt
<function id 2> ::= max | min
```

BOOLEAN EXPRESSION

```
<Boolean expression> ::= <Boolean term> { or <Boolean term>}
<Boolean term> ::= <Boolean factor> { and <Boolean factor>}
<Boolean factor> ::= [not] <Boolean primary>
<Boolean primary> ::= <variable> | <constant> |
                      <relation> | (<Boolean expression>)
<relation> ::= <simple expression> <rel op>
               <simple expression>
<rel op> ::= = | <> | <= | < | >= | >
```

REGULATOR

```
<regulator> ::=
  regulator <regulator ident>;
  { <node declaration>; }
  { <node connection>; }
  end

<node declaration> ::=
  node <node type ident> ( <execution interval> )
    <node ident> { , <node ident> }
<execution interval> ::= <number>

<node connection> ::= <destination> <- <source>
<destination> ::= <node ident>.<variable> |
                  anout.<channel number>
<source> ::= <node ident>.<variable> |
                 anin.<channel number>
<channel number> ::= <integer>
```

Appendix: Intern kod för beskrivning av algoritmerna

Den kod, som produceras av kompilatorn, kan ses som maskinkod till en hypotetisk maskin, kallad H-maskinen. H-maskinen, som simuleras av en särskild process i interpretatorn, är en stackorienterad maskin bestående av tre register, tre minnesareor samt en stack. Minnet är uppdelat i programminne, externt dataminne och internt dataminne. Programminnet innehåller programkoden och dess innehåll ändras inte under programmets exekvering. Det externa dataminnet innehåller de variabler som deklarerats för noden i fråga och det interna dataminnet innehåller de variabler som deklarerats i algortimen. Data kan hämtas från godtyckligt dataminne och läggas överst i stacken (LOAD) och det översta stackelementet kan lagras i godtyckligt dataminne (STORE).

Aritmetiska och logiska operationer göres på de översta stackelementen, som avlägsnas från stacken, och resultaten av operationerna läggs tillbaka överst i stacken.

De tre register, som finns i H-maskinen är programräknaren, PC, som pekar på nästa instruktion i programminnet, instruktionsregistret, I, som innehåller den instruktion som för tillfället exekveras och stackpekaren, SP, som pekar på översta elementet i stacken (stacktoppen).

H-maskinens instruktioner:

Typ 1

Överföring av data mellan stacktopp och dataminne

LOAD adr ext
LOAD adr int
STORE adr ext
STORE adr int

Typ 2

Aritmetiska och logiska operationer på stackens två översta element. Resultat läggs överst i stacken.

ADD
SUB
MULT
DIV
MOD
AND
OR
XOR
LSS
LEQ
EQV
NEQ
GRT
GEQ

Typ 3

Operationer och funktionsberäkningar med stakens översta element som operand. Resultat läggs överst i stacken.

NEG
INT
TAN
ARCTAN
COS
SIN
SQRT
LN
EXP

Typ 4

Kontroll av instruktionsflödet.

JMP	adr .	adr läggs i PC
JPF	adr .	Översta elementet i stacken avlägsnas. Om detta representerar värdet FALSE, läggs adr i PC.
RETURN		Avsluta interpreteringen