CODEN: LUTFD2/(TFRT-7438)/1-136/(1989)

New Tools for Model Development and Simulation

Proceedings of a full-day seminar
Stockholm, October 24, 1989

Sven Erik Mattsson (Ed)

Department of Automatic Control
Lund Institute of Technology
November 1989

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Report

Date of issue
November 1989

Document Number
CODEN:LUTFD2/(TFRT-7438)/1-136/(1989)

Supervisor

Sponsoring organisation
The Swedish Board of Technical Development

Author(s)
Sven Erik Mattsson (Ed)

Title and subtitle
Computer Aided Engineering, CACE
Proceedings of a full-day seminar, Stockholm, October 24, 1989

Abstract

This report contains the proceedings of a full-day seminar held in Stockholm on October 24, 1989. The seminar was arranged by the Swedish National Board for Technical Development, STU, Stockholm and the Department of Automatic Control, Lund Institute of Technology, Lund. The purpose was to present results and experiences from the STU supported research program Computer Aided Control Engineering, CACE. About 85 persons from university and industry attended the seminar.

Key words
Computer Aided Control Engineering

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English and Swedish

Number of pages
136

Recipient's notes

Security classification

Nya verktyg för modellutveckling och simulering

Seminarium 24 oktober 1989
Stockholm

STU-ramprogrammet "Datorbaserade hjälpmedel för utveckling av styrsystem (Computer Aided Control Engineering, CACE)"

Redaktör Sven Erik Mattsson
Inst. för Reglerteknik, Tekniska Högskolan i Lund

STYRELSEN FÖR TEKNISK UTVECKLING
SWEDISH BOARD FOR TECHNICAL DEVELOPMENT

Program och innehållsförteckning

Ordförande: Sven Gunnar Edlund, STFI

Oktober 24, 1989

Simulatorer för processindustrin

Inom området informationsteknologins tillämpningar arbetar STU med ett insatsområde benämnt "Driftutvecklingssystem för processindustrin, DUP". Programmets huvudsakliga inriktning framgår av faktarutan. Inom DUP liksom inom processtyrningsområdet som helhet förväntas simulatorer komma att spela en viktig roll i framtiden. Inom ramen för insatsområdet bedrivs därför simulatorprojekt som har koppling både till basstudier och fallstudier.

Nu är inte användning av simulatorer något som är unikt för processindustrin. Simuleringar har spelat och kommer att spela en viktig roll för en mängd av informationsteknologins tillämpningsområden.

De första simulatorerna

Historiskt sett utvecklades de första simulatorerna som ett hjälpmedel för att lösa dynamiska problem beskrivna med differentialekvationer. Vid lösningen av ekvationerna använde man sig av analoga elektriska kretsar som efterliknade eller simulerade det ursprungliga problemet genom att kretsarnas elektriska samband beskrevs av samma differentialekvationer. Med hjälp av elektroniska förstärkare kunde man bygga maskiner — s k analogimaskiner — som på ett bekvämt och flexibelt sätt kunde användas för att lösa dynamiska problem av mycket stora ordningstal. Genom att analogimaskinerna arbetar tidskontinuerligt och genom parallellbearbetning av problemet är den möjliga problemkomplexiteten helt och hållet kopplad till hårdvarans omfattning.

Digitala datorer

När de digitala datorerna blev vanligare började man använda dessa för att lösa differentialekvationer. Arbetssättet blev då tidsdiskret och sekvensiellt, vilket medförde att det var först med den ökade beräknings- och minneskapaciteten som de digitala datorerna blev det kraftfulla hjälpmedel för simulering som de är i dag. För datorerna utvecklades tidigt en mängd olika typer av beräkningsprogram för t ex konstruktion och dimensionering av processer. Inom en del områden har begreppet simulering därför kommit att användas även för program för olika typer av statiska beräkningar av arbetspunkter och optimeringar. När man i dessa sammanhang studerar de dynamiska problemen talar man då om dynamisk simulering, vilket kan vara något förvirrande för den som har det klassiska simuleringsbegreppet i tankarna.

Värdefullt hjälpmedel

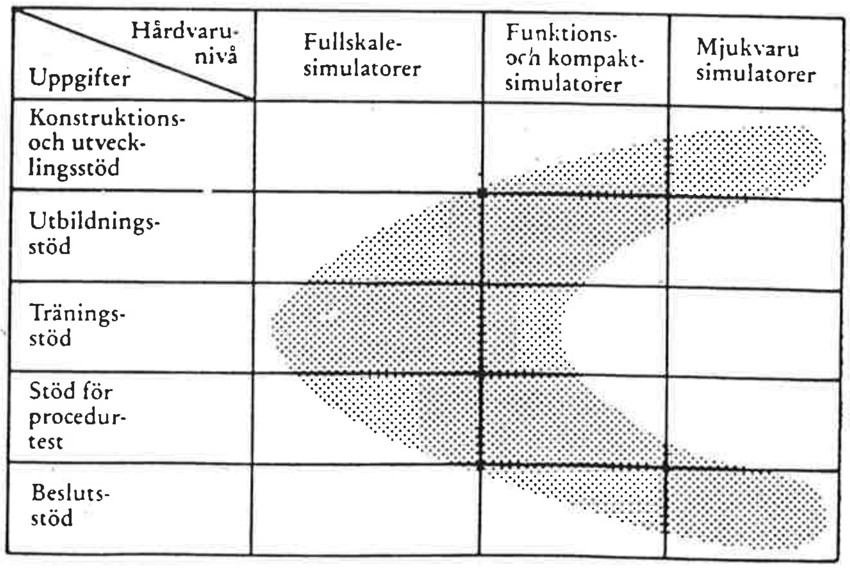
Under flera årtionden har simulatorer varit ett värdefullt hjälpmedel för att prova och utvärdera hur en process eller en systemlösning kommer att fungera utan att systemet behöver byggas eller byggas om till stora kostnader. Simulatorernas inbyggda möjlighet till tidsskalning, så att man kan arbeta i reel tid eller bearbeta problemet långsammare eller snabbare, har varit en extra fördel. På ett tidigt stadium kom simulatorerna även till användning för utbildning och träning av personal vid verksamheter där farliga situationer kunde uppstå vid felaktiga ingrepp. Exempel på sådana tidiga tillämpningar är flyg- och kärnkraftsimulatorer.

Olika aspekter

Principiellt kan man ur processindustrins synvinkel dela in simulatorområdet på flera olika sätt. Följande figur karakteriserar simulatorerna ur två olika aspekter. Horisontellt i figuren har man en indelning efter den hårdvarumässiga uppbyggnaden och likheten med den verkliga processen och processtyrningen. I vertikal led finns en indelning efter simulatorns uppgifter och användningsområden.

Val av simulator

De vanligaste kombinationerna, som har markerats i figuren, kan karakteriseras av följande beskrivningar. Fullskalesimulatorerna, som oftast används för olika typer av träning, kännetecknas av en fullständig instrumentering men kan ha en lägre precision i modellerna än t ex simulatorer för konstruktions- och utvecklingsarbete. Dessa saknar i gengäld oftast den verkliga processtyrningens instrumentering, och hela simulatorn kan bestå av programvara och eventuell datorgrafik som ersätter instrumenteringen (mjukvarusimulator). Andra varianter är dock tänkbara, så bör t ex en simulator för utveckling av människa-maskinsystemet ha en omfattande instrumentering. Här kan fullskalesimulatorer behövas. En annan användning av fullskalesimulatorer är för tester av åtgärdsprocedurer samt start- och stopprocedurer. För grundläggande utbildning kan fullskalesimulatorer vara användbara men här är funktions- och kompaktsimulatorer, som har en förenklad instrumentering och simuleringsmodeller som mera tar fasta på de principiella sambanden, vanligare. Ofta kan man i en funktionssimulator också ha anled-



ning att visa mellanstorheter, som inte är direkt åtkomliga för mätning. Detta för att öka förståelsen.

Beslutsstöd till operatören

Återstår att kommentera den sista raden i figuren, nämligen simulatorer för beslutsstöd till operatörerna. Här handlar det ofta om konsekvenssimulering, varvid operatören i en svårgenomskådlig situation prövar olika tänkbara åtgärder mot simulatorn och får veta konsekvenserna av åtgärderna, innan han genomför dem i verkligheten. Här är en uppsnabbad simulering som sker snabbare än verklig tid en nödvändighet. Precisionen i simuleringsmodellerna måste vara hög och det är viktigt att dessa uppdateras vid förändringar i processen och styrsystemet så att de överensstämmer väl med verkligheten. Kombineras simuleringsmodellerna i beslutsstödet med kunskapsbaserade system eller expertsystem öppnar sig intressanta möjligheter. De arbetsvetenskapliga aspekterna är dock ännu oklara.

Som tidigare nämnts har flera projekt inom DUP-programmet koppling till användning av simulatorer. Vid sidan av Korsnässimulatorn som beskrevs i föregående nummer av IT-AKTUELLT har en utvärdering av simulatoranvändning inom processindustrin utförts av SSPA Maritime Consulting AB på STUs uppdrag. Projektet innefattade förutom en översiktlig genomgång av simulatoranvändningen även studier av fyra simulatorer för olika ändamål. Därutöver har simulatorer och erfarenheter av sådana diskuterats vid olika seminarier som anordnats inom programmet.

Följande citat ur SSPA:s referat av en diskussion av beslutsstödssimulatorer kan ytterligare belysa kraven på dessa simulatorer:

"Användaren måste direkt märka att det går att göra ett bättre jobb och att det lönar sig att använda beslutsstödet".

"En uthållig permanent användning är viktig. Det måste vara lätt att vidareutveckla, bygga ut och komplettera beslutsstödet så att operatören hela tiden kan ha nytta av att använda det"

"Ett beslutsstöd måste användas relativt frekvent för att inte glömmas bort den dag det verkligen behövs."

Kompetensområde

Av ovanstående kommentarer till olika simulatoranvändningar framgår att dessa ställer olika krav på precision eller noggrannhet, på likhet med verklighetens instrumentering och på simuleringens tidsskala. En ytterligare viktig faktor som inte har nämnts uttryckligen, är simulatorns kompetensområde, dvs inom vilket arbetsområde som den ger riktigt resultat med tillfredsställande precision. Alla dessa aspekter gör att det är nödvändigt att man inför en simulatorutveckling klarlägger simulatorns tänkta användningsområde så att den utvecklas med lämpliga prestanda. Det kan vara direkt olämpligt att försöka använda en träningssimulator för beslutsstöd.

Metodutveckling

Simulatoranvändningen är starkt kopplad till lättheten att framställa simuleringsmodeller och att uppdatera dessa. Här kan förbättringar uppnås genom den metodutveckling rörande modellering och simulering som institutionen för reglerteknik vid Lunds tekniska högskola bedriver inom ett av STUs ramprogram CACE (Computer Aided Control Engineering; Datorbaserade hjälpmedel för utveckling av styrsystem). De frågor som bearbetas är bland annat modellsystematik och programspråk för modellerna. Genom lämplig modularisering och standardisering bör man kunna underlätta återanvändningen av utvecklade modeller.

DUPs programledning
Arne Otteblad

DUP — Driftutvecklingssystem för processindustrin — syftar till att tillvarata och utveckla processoperatörens yrkesskicklighet. Målet är också att förbättra kvaliteten och produktjämnheten på produkterna, att bättre utnyttja råvaror och energi samt minska belastningen på miljön. I programmet bedrivs basstudier inom informationsteknologi och arbetsvetenskap samt fallstudier inom branscherna kemi, livsmedel samt massa och papper.

CACE-projektet

STU-ramprogram 1985 – 1989

Datorbaserade hjälpmedel
för utveckling av styrsystem

(Computer Aided Control Engineering)

Huvudaktör

Institutionen för reglerteknik
Lunds Tekniska Högskola

Styrgrupp

Sven Gunnar Edlund, STFI (ordförande)

Arne Otteblad, STU

Karl Eklund, KEAB AB

Claes Källström, SSPA

Erik Sandewall, LiTH

Gustaf Söderlind, ITM

Karl Johan Åström, LTH

Projektledare

Sven Erik Mattsson, LTH

Bakgrund

Dagens CACE-verktyg är användbara
- utför numeriska räkningar
- ritar grafer

men användarna önskar
- bättre användargränssnitt
- stöd för tankeprocessen
- integrerad miljö
  från specifikation till drift av en process.
- utvidgbarhet

Hypotes vid starten

Dagens verktyg designades för 10 – 20 år sedan.

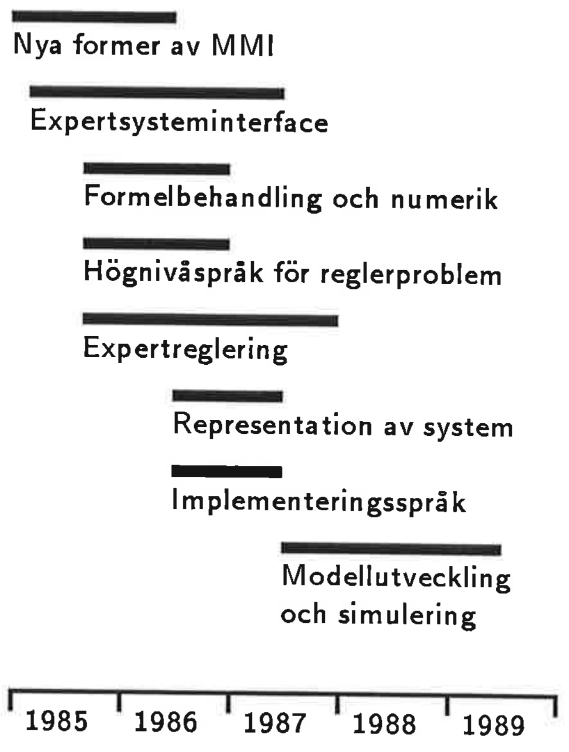
Den enorma utvecklingen inom informationsteknologin
- Arbetsstationer
- Objekt-orienterad programmering
- Databaser
- Grafik och användargränssnitt
- Expertsystemteknik
- Datoralgebra

borde tillåta bättre CACE-verktyg.

Mål

- Undersöka hur de nya landvinningarna inom informationsteknologien kan förbättra CACE-verktygen.
- Utveckla några prototypsystem.
- Etablera internationella kontakter.

Projekt



Publikationer

1 doktorsavhandling
- Karl-Erik Årzén

3 licentiatavhandlingar
- Jan Erik Larsson
- Per Persson
- Bernt Nilsson
- ○ Mats Andersson

8 examensarbeten

5 artiklar
23 konferensbidrag

8 slutrapporter till STU
2 STU-seminarier

Internationella kontakter

5 gästforskare

Den internationella kontaktnätet är stort.
Några exempel:

DK	DTH
NL	Delft, Eindhoven
F	INRIA
GB	SERC, UMIST, Cambridge, Swansea, CEGB, Imperial College
USA	Maryland, RPI, LLNL, Berkeley, Santa Barbara, GE, Mathworks, AT&T Bell Labs

The CACE Project 1989-10-17
Department of Automatic Control
Lund Institute of Technology, Lund, Sweden

Published Papers, Conference Contributions and Reports

ANDERSSON, M. (1989): "An Object-Oriented Modelling Environment," in G. Iazeolla, A. Lehman, H.J. van den Herik (Eds.): *Simulation Methodologies, Languages and Architectures and AI and Graphics for Simulation*, 1989 European Simulation Multiconference, Rome, June 7–9, 1989, The Society for Computer Simulation International, pp. 77–82.

ANDERSSON, M. (1989): "Omola – An Object-Oriented Modelling Language," Report TFRT-7417, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ANDERSSON, M. (1989): "An Object-Oriented Language for Model Representation," IEEE CACSD'89, Tampa, Florida, December 16, 1989.

ANON (1987): "Ramprogram Datorbaserade hjälpmedel för utveckling av styrsystem," *IT-aktuellt*, 2:1987, 5–6.

ÅRZÉN, K-E. (1986): "LISP—A One-Week Course," Report TFRT-7310, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅRZÉN, K-E. (1986): "Expert systems for process control," in Sriram, D. and R. Adey (Eds.): *Applications of Artificial Intelligence in Engineering Problems*, Proc. of 1st Int. Conf. on Applications of AI in Engineering Practice, Southampton University, U.K., April 1986, Springer-Verlag, pp. 1127–1138, Also available as report TFRT-7315, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅRZÉN, K-E. (1986): "Use of Expert Systems in Closed Loop Feedback Control," *Proc. ACC*, Seattle, USA, 1986, pp. 140–145, Also available as report TFRT-7320, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅRZÉN, K-E. (1986): "Kunskapsbaserade Regulatorer," (Knowledge Based Controllers), *SAIS '86*, The Swedish AI Society's Annual Workshop, Linköping, April 24–25, 1986.

ÅRZÉN, K-E. (1986): "Reserapport—AAIEP 1986," Travel Report TFRT-8044, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅRZÉN, K-E. (1987): *Realization of Expert System Based Feedback Control*, Ph.D-thesis TFRT-1029, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅRZÉN, K-E. (1988): "An Architecture for expert system based feedback control," *Preprints of the IFAC Workshop Artificial Intelligence in Real-Time Control*, Swansea, September 20 – 23, 1988, pp. 15–20.

ÅRZÉN, K-E. (1989): "An Architecture for Expert System Based Feedback Control," *Automatica*, Scheduled to appear in November 1989.

ÅSTRÖM, K.J. (1985): "Computer Aided Tools for Control System Design—A perspective," in Jamshidi M. and C.J. Herget (Eds.): *Computer-Aided Control Systems Engineering*, North-Holland, pp. 3–40.

ÅSTRÖM, K.J. (1986): "Auto-Tuning, Adaptation and Smart Control," in Morari and Mc Avoy (Eds.): *Proc Chemical Process Control—CPCIII*, CACHE, Elsevier, pp. 427–466.

ÅSTRÖM, K.J, J.J. ANTON and K-E. ÅRZÉN (1986): "Expert Control," *Automatica*, **22**, No. 3, 277–286.

ÅSTRÖM, K.J. and W. KREUTZER (1986): "System Representations," *Proceedings of the IEEE Control Systems Society Third Symposium on Computer-Aided Control Systems Design (CACSD)*, Arlington, Virginia, September 24–26, 1986, Also available as report TFRT-7330, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ÅSTRÖM, K.J., and S.E. MATTSSON (1987): "High-Level Problem Solving Languages for Computer Aided Control Engineering," Final Report 1987-03-31, STU project 85-4808, STU program: Computer Aided Control Engineering, CACE, Report TFRT-3187, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, D.M. (1986): "Implementation of Graphics for Hibliz," Report TFRT-7328, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, D.M. (1987): "Simplification of Expressions using Prolog," Report TFRT-7364, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, D.M. (1987): "Design and Implementation of a Graphical Front-End," Report TFRT-7367, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, D.M. (1987): "Implementation Languages for CACE Software," Final Report 1987-09-30, STU project 86-4047, STU program: Computer Aided Control Engineering, CACE, Report TFRT-3195, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, D.M. (1988): "Modelling of Control Systems with C++ and PHIGS," *Proceedings of the USENIX C++ Technical Conference*, Denver, Colorado, October 17–20, 1988, pp. 183–192, Also available as report TFRT-7400, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, D.M. (1989): "Experiences of Object-Oriented Development in C++ and InterViews," *Proc. TOOLS'89*, Paris, France, November 13–15, 1989, Also available as report TFRT-7418, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, D.M. (1989): "Scones — An Interactive Block Diagram Editor for Simnon," Report TFRT-7423, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

DENHAM, M.J. (1987): "Knowledge Representation in Systems Modelling," Report TFRT-7365, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQVIST, H. and S.E. MATTSSON (1986): "A Simulator for Dynamical Systems Using Graphics and Equations for Modelling," *Proceedings of the IEEE Control Systems Society Third Symposium on Computer-Aided Control Systems Design (CACSD)*, Arlington, Virginia, September 24–26, 1986, pp. 134–139.

ELMQVIST, H. and S.E. MATTSSON (1989): "Simulator for Dynamical Systems Using Graphics and Equations for Modelling," *IEEE Control Systems Magazine*, **9**, 1, 53–58.

FREDERICK, D.K. (1987): "An Introductory Study of a Window-Based Environment for Simnon on the SUN Workstation," Report TFRT-7366, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

GRANBOM, E. and T. OLSSON (1987): "VISIDYN – Ett program för interaktiv analys av reglersystem," Master thesis TFRT-5375, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

HOLMBERG, U. (1986): "Some MACSYMA Functions for Analysis of Multivariable Linear Systems," Report TFRT-7333, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

HOLMBERG, U., M. LILJA and B. MÅRTENSSON (1986): "Integrating Different Symbolic and Numeric Tools for Linear Algebra and Linear Systems Analysis," Report TFRT-7338, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, Presented at SIAM Conference on Linear Algebra in Signals, Systems and Control, August 12–14, 1986, Boston, Massachusetts, USA.

HOLMBERG, U., M. LILJA and S.E. MATTSSON (1987): "Combination of Symbolic Manipulation and Numerics," Final Report 1987-03-31, STU project 85-4809, STU program: Computer Aided Control Engineering, CACE, Report TFRT-3186, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

HOLMBERG, U., M. LILJA and B. MÅRTENSSON (1988): "Integrating Different Symbolic and Numeric Tools for Linear Algebra and Linear Systems Analysis," in B.N. Datta, C.R. Johnson, M.A. Kaashoek, R.J. Plemmons and E.D. Sontag (Eds.): *Linear Algebra in Signals, Systems, and Control*, Proceedings of the Conference on Linear Algebra in Signals, Systems and Control, Boston, Massachusetts, USA, August 12–14, 1986, SIAM, Philadelphia, pp. 384–400.

JOHANSSON, M. (1988): "Interaktiv plottning av mätdata i flera dimensioner," (Interactive plotting of measurement data in several dimensions), Master thesis TFRT-5390, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

JEPPSSON, U. (1988): "An Evaluation of a PHIGS Implementation for Full Graphics Control Systems," Master thesis TFRT-5389, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

KREUTZER, W. and K.J. ÅSTRÖM (1987): "An Exercise in System Representations," Report TFRT-7369, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

LARSSON, J.E. (1984): "An Expert System Interface for Idpac," Master thesis TFRT-5310, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

LARSSON, J.E. (1989): "Knowledge-Based Frequency Response Analysis," in Larsson, J.E (Ed.): *Proceedings of the SAIS '89 Workshop*, AILU—The AI Group in Lund, Lund University and Lund Institute of Technology.

LARSSON, J.E. and K.J. ÅSTRÖM (1985): "An Expert System Interface for Idpac," Report TFRT-7308, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, Presented at CACSD '85, The 2nd IEEE Control Systems Society Symposium on Computer-Aided Control System Design (CACSD), Santa Barbara, California, March 13–15, 1985.

LARSSON, J.E. and P. PERSSON (1986): "Ett expertsystemsnitt för Idpac," (An Expert System Interface for Idpac), *SAIS '86*, The Swedish AI Society's Annual Workshop, Linköping, April 24–25, 1986.

LARSSON, J.E. and P. PERSSON (1986): "Knowledge Representation by Scripts in an Expert Interface," *Proceedings of the 1986 American Control Conference*, Seattle, June 1986, Also available as report TFRT-7332, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

LARSSON, J.E. and P. PERSSON (1987): "An Expert System Interface for Idpac," Lic Tech thesis TFRT-3184, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

LARSSON, J.E. and P. PERSSON (1987): "Ett intelligent gränssnitt för systemidentifiering," (An Intelligent Interface for System Identification), *Proceedings of the SAIS'87 Workshop*, The Swedish AI Society's Annual Workshop, Uppsala, May 18–19, 1987.

LARSSON, J.E. and P. PERSSON (1987): "Experiments with an Expert System Interface," Final Report 1987-09-30, STU project 85-3042, STU program: Computer Aided Control Engineering, CACE, Report TFRT-3196, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

LARSSON, J.E. and P. PERSSON (1987): "The (ihs) Reference Manual," Report TFRT-7341, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

LARSSON, J.E. and P. PERSSON (1987): "A Knowledge Database for System Identification," Report TFRT-7342, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

LARSSON, J.E. and P. PERSSON (1988): "An Intelligent Help System for Idpac," *Proceedings of ECAI-88*, European Conference on Artificial Intelligence, Munich, August 1 – 5, 1988, pp. 119–123.

LARSSON, J.E. and P. PERSSON (1988): "The knowledge database used in an expert system interface for Idpac," *Preprints of the IFAC Workshop Artificial Intelligence in Real-Time Control*, Swansea, September 20 – 23, 1988, pp. 107–112.

LILJA M. (1986): "Some SISO Transfer Function Facilities in CTRL-C," Report TFRT-7325, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. (1985): "New Forms of Man-Machine Interaction," Status Report 1985-09-30, STU project 84-5069, STU program: Computer Aided Control Engineering, CACE, Report TFRT-7293, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. (1986): "On Differential/Algebraic Systems," Report TFRT-7327, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. (Ed.) (1987): "Computer Aided Control Engineering, CACE," Proceedings of a full-day seminar, Stockholm, March 25, 1987 Report TFRT-7359, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. (Ed.) (1987): "Programplan för ramprogrammet Datorbaserade hjälpmedel för utveckling av styrsystem (Computer Aided Control Engineering, CACE)," Final Report TFRT-3193, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. (1987): "Hibliz ger enkel och klar modellbeskrivning på varje nivå," *IT-aktuellt*, 2:1987, p. 11.

MATTSSON, S.E. (1987): "Representation and Visualization of Systems and Their Behaviour," Final Report 1987-09-30, STU project 86-4049, STU program: Computer Aided Control Engineering, CACE, Report TFRT-3194, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. (1988): "On Model Structuring Concepts," *Preprints of the 4th IFAC Symposium on Computer-Aided Design in Control Systems (CADCS)*, August 23–25 1988, P.R. China, pp. 269–274.

MATTSSON, S.E. (1989): "On Modelling and Differential/Algebraic Systems," *Simulation*, **52**, No. 1, 24–32.

MATTSSON, S.E. (1989): "Modelling of Interactions between Submodels," in G. Iazeolla, A. Lehman, H.J. van den Herik (Eds.): *Simulation Methodologies, Languages and Architectures and AI and Graphics for Simulation*, 1989 European Simulation Multiconference, Rome, June 7–9, 1989, The Society for Computer Simulation International, pp. 63–68.

MATTSSON, S.E. (1989): "Concepts Supporting Reuse of Models," *Proceedings of Building Simulation '89*, Vancouver, June 23–24, 1989, The International Building Performance Simulation Association, pp. 175–180.

MATTSSON, S.E. and M. ANDERSSON (1989): "A Kernel for System Representation," Report TFRT-7429, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. and M. ANDERSSON (1989): "An Environment for Model Development and Simulation," Final Report 1989-09-30, STU projects 87-2503 and 87-2425 STU program: Computer Aided Control Engineering, CACE, Report TFRT-3205, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. and K.J. ÅSTRÖM (1986): "The CACE Project— Steering Committee Meeting 2," Report TFRT-7321, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. and K.J. ÅSTRÖM (1986): "The CACE Project— Steering Committee Meeting 3," Report TFRT-7322, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. and K.J. ÅSTRÖM (1987): "The CACE Project— Steering Committee Meeting 4," Report TFRT-7343, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. and K.J. ÅSTRÖM (1988): "The CACE Project— Steering Committee Meeting, 1987-11-25," Report TFRT-7375, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. (1988): "The CACE Project— Steering Committee Meeting, 1988-06-01," Report TFRT-7395, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. (1989): "The CACE Project— Steering Committee Meeting, 1988-11-23," Report TFRT-7412, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E., K.J. ÅSTRÖM, D.M. BRÜCK and T. SCHÖNTHAL (1987): "A Trip to the University College Swansea and the Central Electric Generating Board, Gloucester," Travel Report TFRT-8045, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E., H. ELMQVIST and D.M. BRÜCK (1986): "New Forms of Man-Machine Interaction," Final Report 1986-09-30, STU project 84-5069, STU program: Computer Aided Control Engineering, CACE, Report TFRT-3181, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MÅRTENSSON, B. (1986): "CODEGEN—Automatic Simnon Code Generator for Multivariable Linear Systems," Report TFRT-7323, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MÅRTENSSON, B. (1986): "Automatic TEX Code Generation from Macsyma and CTRL-C," Report TFRT-7334, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MÅRTENSSON, B. (1986): "Hcopy2PS-Hcopy to PostScript filter," Report TFRT-7335, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MÅRTENSSON, B. (1986): "Experiences With Computerized Document Preparation Tools," Report TFRT-7336, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MÅRTENSSON, B. (1987): "Pascal and Fortran Systems in VAX/VMS Simnon – A Cookbook," Report TFRT-7351, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MÅRTENSSON, B. (1987): "Documentation of MacEQ2TEX, DVILW, and Hcopy2PS," Report TFRT-7352, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, B. (1986): "LQG-I/O—A CTRL-C library for LQG design," Report TFRT-7329, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, A. (1988): "Object-oriented Graphics for the Future Instrument Panel," Master thesis TFRT-5382, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, B. (1987): "Experiences of Describing a Distillation Column in Some Modelling Languages," Report TFRT-7362, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, B. (1989): "Structured Chemical Process Modelling — An object oriented approach," Lic Tech thesis TFRT-3203, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, B. (1989): "Structured Modelling of Chemical Processes with Control Systems," Annual AIChE Meeting 1989, San Francisco, Nov 5–10, 1989.

NILSSON, B., S.E. MATTSSON and M. ANDERSSON (1989): "Tools for Model Development and Simulation," in Larsson, J.E (Ed.): *Proceedings of the SAIS '89 Workshop*, AILU—The AI Group in Lund, Lund University and Lund Institute of Technology.

VALLINDER, P.A. (1988): "Some Methods for Tearing of Differential/Algebraic Systems," Master thesis TFRT-5384, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

En objektorienterad miljö för modellutveckling

Sven Erik Mattsson och Mats Andersson

Institutionen för reglerteknik
Lunds Tekniska Högskola

Innehåll

Varför behövs modeller?

För att uppfylla
- lönsamhetskrav
- kvalitets- och prestandakrav
- säkerhetskrav

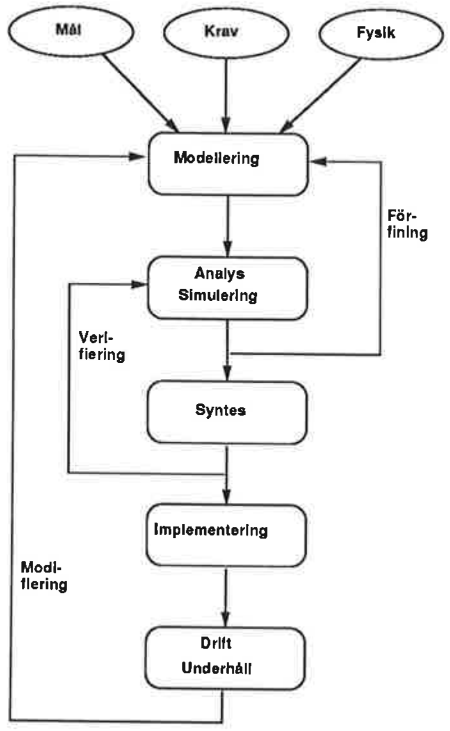
behövs kunskap.

Modeller är formaliserad kunskap

Kunskap

- "Göra som man alltid gjort"
- Experimentera och pröva sig fram
  - prototyper
  - fullskala
- Matematiska modeller och metoder

Modelleringsprocess



Användningsområden

Projektering:
1. Analys
2. Design
3. Validering
4. Utbildnings- och träningssimulatorer
5. Dokumentation

Drift:
1. Produktionsplanering
2. Styrning och reglering
3. Driftsoptimering
4. Operatörsstöd
5. Övervakning och diagnos
6. Felanalys
7. Underhåll

Alla matematiska metoder behöver modeller.

Drivande krafter

Konkurrens kräver kvalitet och prestanda
- Testa många lösningar
- Svårt att ställa in mer än 3 parametrar manuellt

Spara energi, råmaterial och miljö
$\Rightarrow$ Kopplade och återkopplade system
$\Rightarrow$ Komplexare system

Säkerhet
- Experiment kan vara farliga
- Validera beteende i extrema situationer
- Träna extrema situationer
- Övervakning och felanalys
- Myndighetskrav

Kostnader
- Dyrt med experiment och misslyckanden
- Inte störa driften

En integrerad miljö

$\star$ Traditionell modellering

$\star$ En integrerad modelleringsmiljö

$\star$ Arkitektur

$\star$ Återanvändning

$\star$ Modellabstraktion

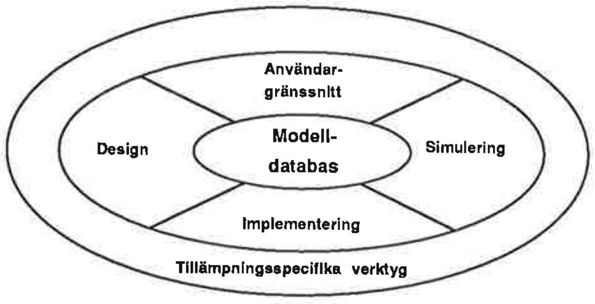
$\star$ Representation / presentation

Traditionell modellering

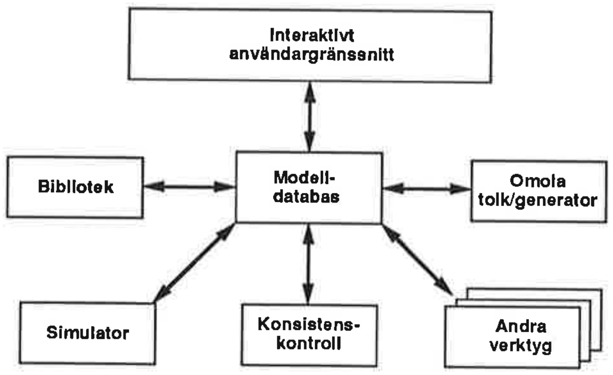
- Programmeringsspråk (t.ex FORTRAN)
  - Låg nivå, fjärran från fysik och verklighet.
  - Modell och simulator blandade.
  - Svårt att underhålla modellen.
- CSSL, 1967
- Verktygsberoende modeller.
- Slutna moduler.

Vårt mål

- Enhetlig, generell representation av modeller.
- Integrerad miljö för modellutveckling, simulering och design.
- Modelldatabas.
- Anpassningsbara användargränssnitt.



Arkitektur



Återanvändning

Modeller kan återanvändas

- som delar i andra modeller
- av andra personer
- för andra syften

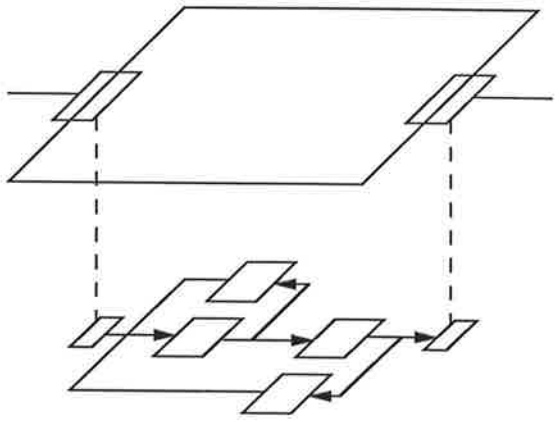
Återanvändning ställer krav

- Modelleringsspråket
  - generalitet
  - strukturering
  - modularitet
- Modellbyggaren måste
  - strukturera
  - dokumentera
- Verktygen
  - hantera strukturerade modeller
  - bibliotek
  - konsistenskontroll

Modellabstraktion

Inkapsling för att hantera komplexa system.
En modell beskrivs av

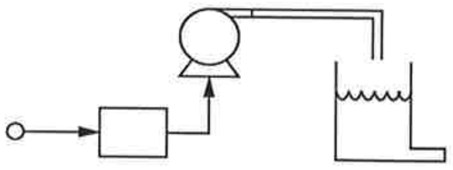
- Gränssnitt
- Realisering

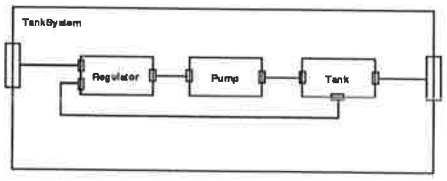


Representation – presentation

Gemensam representation, olika presentation.

```
(Tanksystem ((input FlowTerminal)
            (output FlowTerminal))
           ((regulator PID)(pump Pumpmodel)
            (tank Tankmodel))
           ((input (regulator r) ...
)))
```

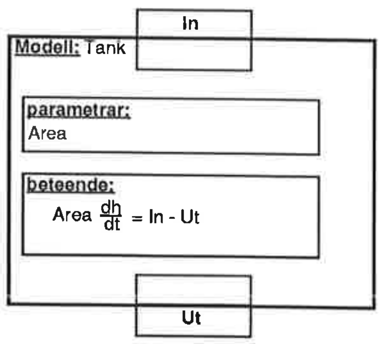




Modellkomponenter

En modell består av

1. Terminaler
2. Parametrar
3. Realiseringar



Hierarkisk modularisering

Realiseringar kan vara
- Strukturerade
  ihopkopplade delmodeller
- Primitiva
  ekvationer

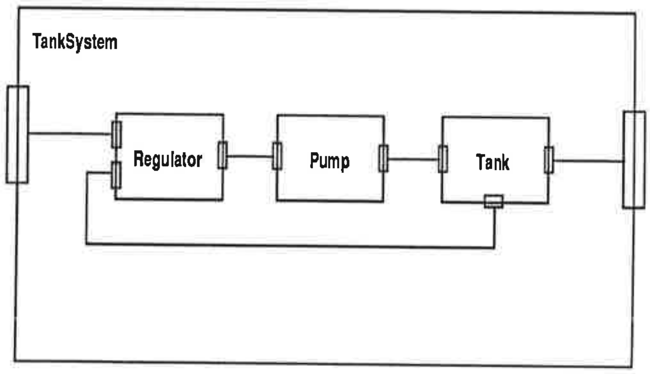
En modell kan ha flera realiseringar
- Versioner
- Tillståndsmoder

Terminaler

Terminaler kan vara strukturerade:
- Enkla – en storhet
- Strukturerade
  - elektrisk ledning: spänning och ström
  - hål: massflöde, tryck, temperatur
  - axel: läge, kraft, moment
  - kablar har flera trådar
  - flöden kan bestå av flera komponenter

Modellering av växelverkan

Göres genom att koppla terminaler:
- Likhet eller nollsumma
- Delmodeller för komplex växelverkan
- Blockdiagram



Modellmodularisering

Komponentbaserade modeller är en möjlighet:
- Bygga modellen som systemet
- Gemensam bas
- Modeller från komponenttillverkarna
- Möjligt att förutse fysiska kopplingar
- Terminalbibliotek

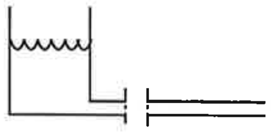
Konsistenskontroll av kopplingar

En enkel terminal har attributen
- Riktning: odefinierat, in, ut
- Storhet – ISO 31
- Enhet
- Undre och övre gräns
- Variabilitet:
  tidsvariabel, parameter, konstant
- Kausalitet: okänd, läs, skriv

Om ej tidsvariabel
- Propagering av parametervärden
- Värden måste vara konsistenta

Ett exempel

Hål och rörändar



	Storhet	Enhet	
Diameter	längd	mm	parameter
Medium			
Flöde	massflöde	ml/s	in
Tryck	tryck	Pa	
Temp	temperatur	K	

Vektorer för att beskriva multimediaflöden.

Terminaler

Kan var delvis ospecificerade:
1. Typ av komponenter
2. Antal komponenter
3. Storhet, enhet mm.

för att tillåta
1. Generiska modeller
2. Abstraktion – Top-down
3. Automatisk deklaration – Bottom-up

Terminaltypbegrepp

Vilka terminaler kan kopplas samman?

Klassiska typbegrepp
1. Strukturekvivalens
   för svag
2. Namnekvivalens
   för snäv

Vårt terminalbegrepp tillåter explicit, redundant information.

Ekvationer

Ekvationer är en naturlig form:
- mass- och energibalanser
- rörelseekvationer mm
- DAE: $g(t, \dot{x}, x) = 0$

Idag

Dagens simuleringsprogram löser

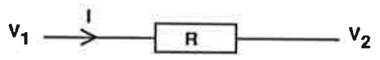
$$\frac{dx}{dt} = f(t, x), \quad x(t_0) = x_0.$$

om användaren definierar en procedur som beräknar derivatorna, $f(t, x)$.

Nackdelar
- användaren måste manipulera ekvationerna
- In/Utsignaler – Beräkningskausalitet

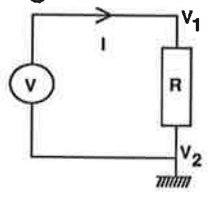
In- eller Utsignaler?

Gör en delmodell för ett motstånd



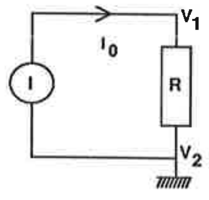
1 Parameter: R
3 Variabler: V_1, V_2, I
1 Ekvation: $V_1 - V_2 = RI$ (Ohms lag)
$\Rightarrow$ Modellen måste ha 2 insignaler

Fall 1: Kopplad till en spänningskälla



Insignaler: $V_1 = V$ och $V_2 = 0$
Utsignal: $I = (V_1 - V_2)/R$

Fall 2: Kopplad till en strömkälla

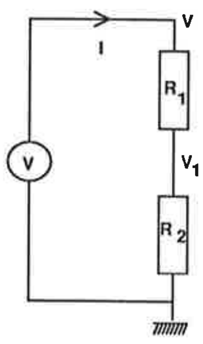


Insignaler: $I = I_0$ och $V_2 = 0$
Utsignal: $V_1 = V_2 + RI$

Vi kan inte i förväg säga vad som skall vara insignaler.

Tyvärr är situationen ännu värre.

Fall 3: Två motstånd i serie



Oberoende av hur vi väljer in/utsignaler för motståndsmodellerna måste ett ekvationssystem lösas;

$$V_1 + R_1 I = V$$
$$V_1 - R_2 I = 0$$

Deklarativa modeller

En modell skall
- beskriva fakta och relationer
- inte vara en beräkningsprocedur

Fördelar:
- Närmre fysiken
- Ger bättre dokumentation
- Tillåter konsistenstest
- Stöder återanvändning
  - delmodeller
  - olika beräkningar

Modeller på ekvationsform

Möjliggör automatisk generering av:
- Effektiv simuleringskod
- Kod för att beräkna arbetspunkter
- Linjära modeller
- Beskrivningar till andra program
- Regulatorkod

Omola

⋆ Allmänt om Omola

⋆ Klasser och ärvning

⋆ Modeller, terminaler, mm

⋆ Omola är generellt

⋆ Utvidgningar

Omola = Object-oriented MOdelling Language

Ett språk för att beskriva modellers
- struktur
- beteende
- gränssnitt
- relationer

Omola är ett textformat för modelldatabasen:

Omolakod $\Longleftrightarrow$ Modelldatabas

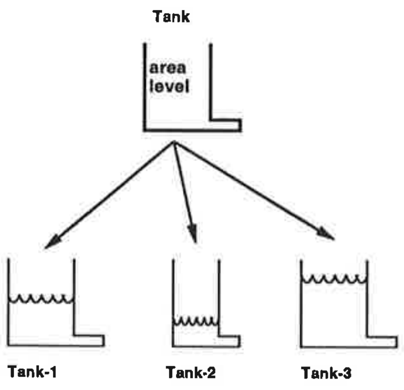
Idéer och begrepp från objektorienterad programmering.

Klasser

En klass beskriver en grupp objekt med liknande egenskaper.

Modellobjekt (modeller, terminaler, realiseringar ...) representeras av klasser med attribut.

Attribut kan vara variabler eller klasser.

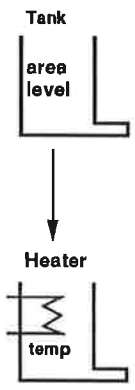


Ärvning

Relation mellan klasser:
superklass — subklass

En subklass ärver superklassens attribut.

En subklass är en specialisering av sin superklass.



En modell i Omola

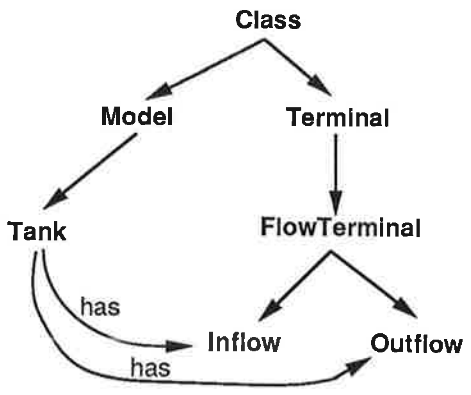
```
Tank ISA Model WITH
  parameters:
    area TYPE real;
  terminals:
    Inflow  ISA FlowTerminal;
    Outflow ISA FlowTerminal
END
```

En klass består av attribut uppdelade i kategorier.

Klassrelationer

Det finns två slags relationer mellan klasser:

- subklass — superklass (isa)
- komponent — ägare (has)



Realiseringar

Exempel på primitiv realisering:

```
Tank ISA Model WITH
    .
    .
 realization:
  TankBehaviour ISA SetOfDAE WITH
   equations:
    outflow = k*sqrt(level);
    area*dot(level) = inflow - outflow;
  END;
END;
```

Exempel på strukturerad realisering:

```
TankSystem ISA Model WITH
    .
    .
 realization:
  TankStruct ISA Structure WITH
   submodels:
    Tank1 ISA Tank;
    Tank2 ISA Tank;
   connections:
    Tank1.Outflow AT Tank2.Inflow;
    ...
  END;
END;
```

Terminaler

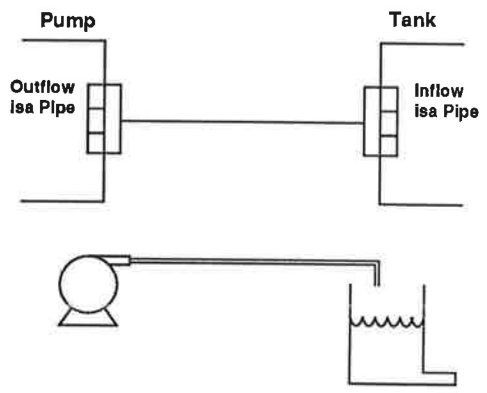
Enkla terminaler:

```
SimpleTerminal ISA Terminal WITH
 attributes:
  value TYPE Real;
  quantity TYPE Quantity;
  unit TYPE String;
  direction TYPE (Across, In, Out);
  .
  .
END;
```

```
FlowTerminal ISA SimpleTerminal WITH
 quantity = MassFlow;
 unit     = "kg/s";
 direction = In;
END;
```

Strukturerad terminal:

```
Pipe ISA RecordTerminal WITH
 components:
  FlowRate ISA FlowTerminal;
  Pressure ISA PressureTerm;
  Temp ISA TempTerminal;
END;
```



Parametrar och bivillkor

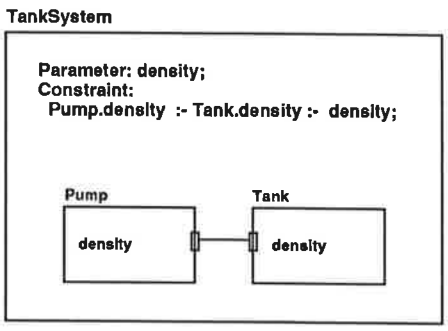
Parametrar ska kunna låsas

- till ett konstant värde,
- till andra parametrar.

Specialisering:

```
Tank ISA Model WITH
 parameters:
  density TYPE Real;
END;

WaterTank ISA Tank WITH
 density = 1;
END;
```

Bivillkor:



Omola är generellt

Omola är konstruerat för att ha

- enkel syntax,
- få koncept,
- flexibel semantik.

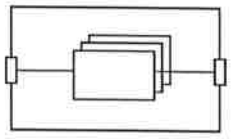
Enkelt att anpassa Omola för nya verktyg och nya typer av modeller genom:

- nya attribut,
- nya kategorier.

Utvidgningar

- Samplade system
- Diskreta händelser
- Procedurer

Händelser kan byta realisering:



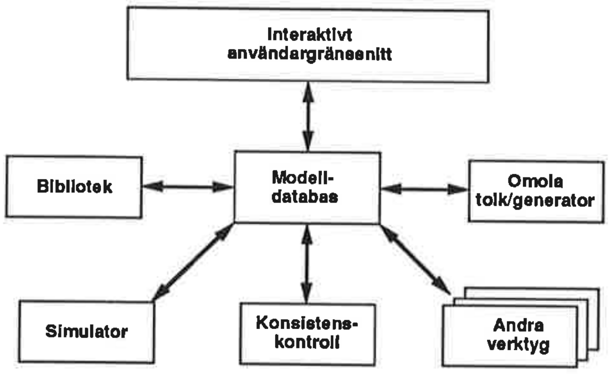
```
WHEN temp > highlimit SELECT high_temp_model;
WHEN temp < lowlimit SELECT low_temp_model;
```

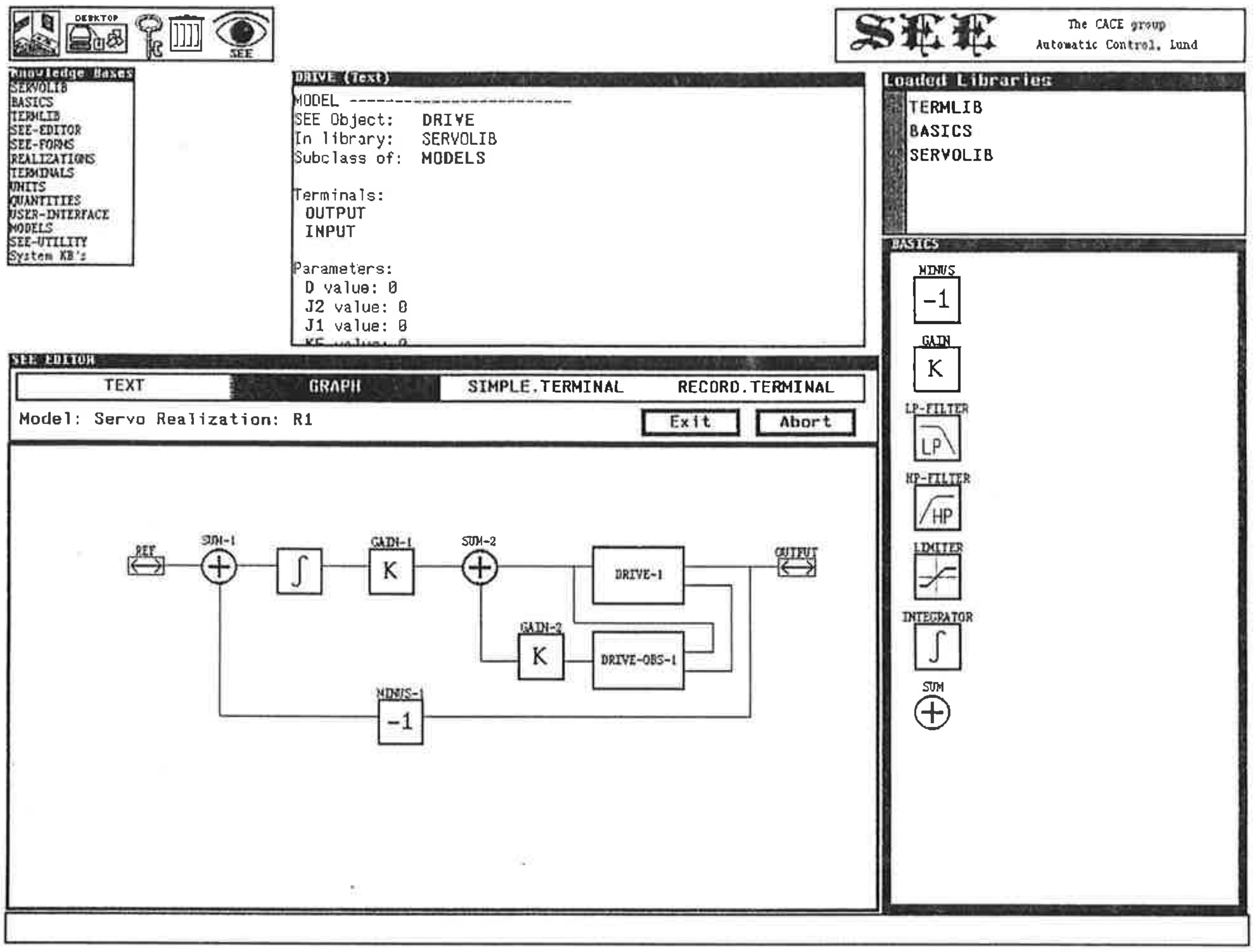
Exekvera en procedur:

```
WHEN level > alarm_level DO
  .
  .
END
```

Prototypen: SEE

SEE är baserad på KEE och CommonLisp.

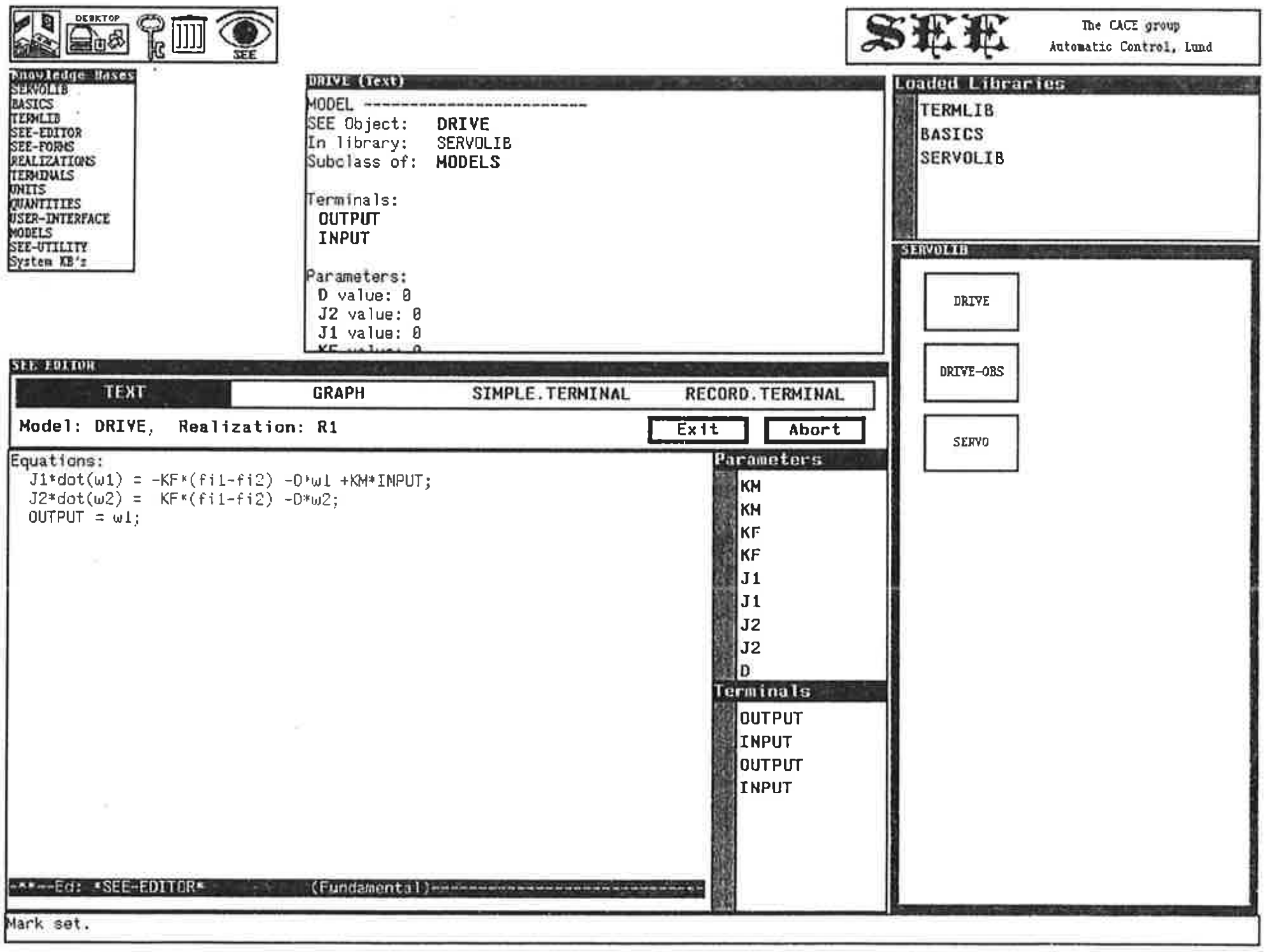




DRIVE (Text)

```
MODEL -------------------------
SEE Object:   DRIVE
In library:   SERVOLIB
Subclass of:  MODELS

Terminals:
 OUTPUT
 INPUT

Parameters:
 D value: 0
 J2 value: 0
 J1 value: 0
 KF value: 0
```

Model: Servo Realization: R1



DRIVE (Text)

```
MODEL -------------------------
SEE Object:   DRIVE
In library:   SERVOLIB
Subclass of:  MODELS

Terminals:
 OUTPUT
 INPUT

Parameters:
 D value: 0
 J2 value: 0
 J1 value: 0
 KF value: 0
```

Model: DRIVE, Realization: R1

```
Equations:
  J1*dot(w1) = -KF*(fi1-fi2) -D*w1 +KM*INPUT;
  J2*dot(w2) =  KF*(fi1-fi2) -D*w2;
  OUTPUT = w1;
```

```
-**--Ed: *SEE-EDITOR*          (Fundamental)------------------
Mark set.
```

Resultat

Kärna för modellrepresentation
- Designförslag
- Prototyp i Common Lisp och KEE
- ○ Central modelldatabas i en miljö med ingenjörsverktyg.
- ○ Integrerade och användaranpassade gränssnitt och verktyg.

Tillämpningsprojekt:
Modellering av kemiska processer.

Ny standard för modellrepresentation behövs.

Kärnans modellrepresentation

- Deklarativa och ekvationsbaserade beteendebeskrivningar gör modellerna användbara för olika uppgifter.
- Hierarkiska modeller med väldefinierade gränssnitt: terminaler och parametrar.
- Terminalattribut för automatisk konsistenskontroll av kopplingar.
- Multipla beteendebeskrivningar; modellvarianter och alternativt beteende.
- Objektorienterad representation; klasser med ärvning underlättar återanvändning och inkrementell modellutveckling.
- Den interna representationen bevarar modellstrukturen.

Utveckling och tekniköverföring

- Konferensbidrag och artiklar
- Programkomponenter
- Tillämpningsprojekt
- Standarder

Programkomponenter

- bra sätt att sprida idéer och metoder
- behövs i tillämpningsprojekten
- prototyp i Common Lisp och KEE

Kärna för modellutveckling och simulering

Implementera en kärna för modellutveckling och simulering i C++.

STU-projekt 1989 – 1981.

Inte ett universitets uppgift att utveckla och marknadsföra kommersiella programkomponenter.

Tjäna som en bas för någon annan att utveckla kommersiella produkter.

Nya verktyg för användargränssnitt på väg.

Tillämpningsprojekt

Mål
- sprida resultaten
- lägga grunden för en bred användning
- få återkoppling från användare
- utveckla modellbibliotek
- utveckla tillämpningsspecifika verktyg och användargränssnitt

Beslutade projekt
- IT4: steritherm-process
- DUP: sockerkristallisation

Projektförslag välkomnas!

Standardisering av modellspråk

IMACS
International Association for Mathematics and Computers in Simulation

SCS
The Society for Computer Simulation

ISO
- Standardförslag till datarepresentation

IFAC
International Federation of Automatic Control
- WG on "Guidelines for CACSD-software"
- Förslag för linjära system

IBPSA
International Building Performance Simulation Association
- Förslag till neutralt format

Möjliga fortsättningar

- Diskreta händelser i kontinuerliga modeller
- Presentationsformer för reglersystem och dess struktur
- Bibliotekshjälpmedel
- Parameterskattningsverktyg
- Symbolisk manipulering och analys av modeller

CODEN: LUTFD2/(TFRT-3205)/1-030/(1989)

An Environment for Model Development and Simulation

Sven Erik Mattsson
Mats Andersson

STU projects 87-02503, 87-02425

Department of Automatic Control
Lund Institute of Technology
September 1989

Department of Automatic Control **Lund Institute of Technology** P.O. Box 118 S-221 00 Lund Sweden	*Document name* Final Report *Date of issue* 1989-09-30 *Document Number* CODEN:LUTFD2/(TFRT-3205)/1-030/(1989)
Author(s) Sven Erik Mattsson Mats Andersson	*Supervisor* *Sponsoring organisation* The National Swedish Board of Technical Development (STU contracts 87-2503, 87-2425)

Title and subtitle

An Environment for Model Development and Simulation

Abstract

This is a final report for the project "Tools for model development and simulation" (STU projects 87-02503, 87-02425) carried out in the period July 1987 to June 1989. The project is the last part of the STU supported research program "Datorbaserade hjälpmedel för utveckling av styrsystem (Computer Aided Control Engineering, CACE)", which started in the end of 1984. The main result is a proposal for a kernel for model representation. The kernel may serve as a central model data base in an integrated environment for model development and simulation. The CSSL definition from 1967 has had a profound impact on simulation and has served very well for over 20 years. It is perhaps now time to capitalize on the enormous development of information technology and reconsider the foundations of model representation. The proposal is a modest effort in this direction. If we could agree upon a common set of ideas we may lay the foundation to a new standard. The proposed kernel supports a modularized and object oriented representation of models to allow flexible and safe reuse of model components. The model developer may supply extra information which is used for automatic consistency analysis to check for unintended abuse of models. The kernel can allow any logical and mathematical framework such as differential-algebraic equations, difference equations, etc. to describe behaviour, but a basic idea is that behaviour descriptions should be declarative and equation based. The kernel allows integration of different customized user interfaces. A prototype of the kernel is implemented in Common Lisp and KEE. A new STU-supported project to implement the kernel in C++ has been started. The project has also included an application study focusing on modelling of chemical processes.

Key words

Computer aided control engineering; computer aided system design; modelling; simulation languages; hierarchical systems; system representations; data structures.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title		*ISBN*
Language English	*Number of pages* 30	*Recipient's notes*
Security classification		

The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Telex: 33248 lubbis lund.

Contents

1. Introduction

This is a final report for the project "Tools for model development and simulation" (STU projects 87-02503, 87-02425) carried out in the period July 1987 to June 1989. The project is the last part of the STU supported research program "Datorbaserade hjälpmedel för utveckling av styrsystem (Computer Aided Control Engineering, CACE)", which started in the end of 1984.

Automation and advanced control are of strategic importance for the Swedish industry. There are examples in the whole range from traditional process industry and power generation to aero- and astronautics. To be able to develop and operate high performance systems, computer based tools for model development, simulation, analysis, design, validation, operator training, production planning, operator support, supervision, fault diagnosis etc. are needed.

Today's CACE tools have proved to be useful. However, they were designed 10–20 years ago. The computers had then moderate computing capacity and primitive hardware for graphical input and output. The main function of the tools is to perform extensive numeric calculations and present the results in the form of simple plots. The users want to have tools that better support their needs: user interfaces which support their way of thinking, integrated environments supporting all phases from specification and design to operation and maintenance etc. The enormous development of the information technology (workstations, object-oriented programming, computer graphics, artificial intelligence, expert system techniques, computer algebra etc.) has opened possibilities to improve the CACE tools significantly. The goal of the CACE project has been to

1. investigate how the CACE tools can be improved
2. develop prototype tools
3. establish international contacts

In the first phase of the CACE project a number of pilot projects investigated some ideas and the potential of computer graphics, computer algebra and expert system techniques. Prototype tools, which can demonstrate ideas and principles were also developed. These projects showed that it indeed is possible to improve the tools. A summary can be found in Mattsson (1987).

For the last phase of the CACE project it was decided to focus on tools for model development and simulation (Mattsson, 1987). The motives were:

1. It is of importance for all kinds of engineering.
2. It contains most of the important issues for CACE.
3. It fitted well in the international collaboration.

An important conclusion from the pilot projects was that model representation is a critical issue. The system concept is fundamental in control engineering, but today's tools have only primitive representations, which do not support the users' perception of systems. Furthermore, a common basic representation is needed to make the CACE tools integrated.

The work in last phase of the CACE project has largely followed the research program. A major result is a design proposal for a kernel for model development and simulation. The proposal may be of interest for all users of models. A basic idea is to support reuse of models so that models can be used for different tasks and so that it easy to modify a model to describe a similar system, since it is difficult and laborious to develop new models. By a kernel we mean the routines to manipulate the internal representation. In our design there is a clear separation between user interface, internal representation of data and models and processing tools. This separation makes the design more flexible and allows customized user interfaces. The kernel can be viewed as a central model data base in an integrated environment for model development, simulation, analysis, design, documentation etc. A prototype implementation of the kernel as well as a user interface has been written in Common Lisp and KEE. The project has also comprised an application study focusing on modelling of chemical processes to get some evaluation of the ideas.

This report is organized as follows. In Chapter 2 motives for supporting model development are given. Chapter 3 outlines basic ideas and our approach to support model development. Chapters 4 and 5 describe the kernel in some detail. Chapter 6 is about user interfaces and Chapter 7 is about the application study. Chapter 8 contains the conclusions. Appendices A – C list published papers, conference contributions, other reports and external lectures given by CACE group members.

2. Models Are Essential

The reason for supporting model development is that

1. models are essential in all kinds of engineering and
2. model development is difficult and time consuming.

It is a well-known fact that it is difficult and time-consuming to develop a new model and we will discuss approaches to support model development in the next section. Let us now motivate why models are needed.

What are the uses of models?

Models are useful in all phases of a systems life from design to operation and maintenance. The designer can use a model to simulate and to analyse the behaviour to learn about the system and to get insight in its behaviour and to validate his design. He can try various system architectures or configurations to make the best choice. He can use optimization tools to tune system parameters. Models are needed in simulators for education and training. Computer based tools for production planning, online optimization, operator support, supervision and failure analysis need models of the system.

Note that modelling and simulation are closely connected to each other. To simulate you need a model. Realistic models are typically non-linear, which implies that it is difficult to analyse a behaviour in other ways than through simulations. However, with a modelling language clearly separated from calculation and simulation issues, models can be used in a more general context for process documentation and to preserve design knowledge.

Why are models needed?

All mathematical methods need some kind of model of the system under consideration. If we do not want to use mathematical methods and models when making a new design we have to make trial-and-error experiments on real equipment. It may be unfeasible to make experiments on real equipment for complexity, performance, safety and economic reasons. First, the system to be designed may have to be so complex that it is impossible to come up with any reasonable design from trial-and-error experiments. Second, to achieve high performance the system must be optimized, but it is in practice impossible to tune more than three coupled parameters by trial-and-error. Third, safety regulations may forbid real experiments, or require validation of the design for extreme and emergency conditions and it may be dangerous or impossible to perform this validation by real experiments. Fourth, real experiments are often expensive and time consuming to perform. Furthermore, when redesigning a plant, it may not be allowed to disturb the operation of the existing plant.

Power generation, aero- and astronautics are typical areas where advanced mathematical methods have been used for a long time to handle complexity, performance and safety issues.

Fierce competition is an important force to use advanced mathematical methods to make better and cheaper designs, and to use computer based tools for production planning, online optimization, operator support, supervision and failure analysis to increase productivity and quality and to decrease production and maintenance costs.

Requirements on saving energy and raw material as well as avoiding environmental pollution make the designs more complex, since the system must contain recirculation loops to win back energy and material. Recirculation loops introduce interactions between various parts of the process implying that it is impossible to design and to operate them independently of of each other.

More specific motives for using advanced mathematical methods for design and in particular control design can be found in Anon (1987) and Fleming (1988). The US Department of Defense has picked simulation and modelling technology as one of 22 critical technologies, since it can reduce design and production costs, improve performance and maintenance, train personnel. Simulators for education and training have been used for a long time in power generation, aero- and astronautics. The interest from other industry areas to use simulators for education, training and operator support is large today. STU has a special research program DUP to investigate how process operators' tasks can be supported by computer based tools. A large part of this program is devoted to simulators.

3. Support of Model Development

In this chapter we will first indicate requirements on concepts and tools for model development and then outline our approach.

3.1 Requirements

Since models are important and since it is difficult to develop new models, a basic question is how computer based tools can support model development?

Reuse

The best way is of course to be able to provide the user with the desired model directly and automatically. This implies model libraries and reuse of models. There are three facets of reuse:

1. *Various purposes or calculations.*
   Models are needed in all mathematical methods and it should be possible to use a model for various purposes without having to recode it manually.
2. *Similar systems.*
   It should be easy to adapt a model to describe a similar system.
3. *Different users.*
   The user interface should preferable be customized and adapted to understand and use the user's concepts and terminology.

New models

A model can be developed using first principles or by analysing measured data. Our project have mainly focused on the first approach.

When developing a new model, decomposition is needed to handle complexity. It should also be possible to extract and reuse parts of existing models. There should be tools that tune model parameters from measured data.

3.2 Basic ideas

Our proposal is based on four main ideas

1. declarative models
2. structured models
3. automatic consistency checking
4. customized user interfaces

Declarative models

Models developed to be used in one package today cannot be used in another package without additional work. Unfortunately, much "model development" work of today consists of manual recoding or implementation of adapters.

An obvious reason is of course that there is no common agreement on the representation of models.

Another maybe less obvious reason is that the representations used in most of today's CACSD and simulation tools are too specialized and of too low a level to allow reuse of models for other tasks than simulation. Today's most used languages for continuous simulation (ACSL, CSMP, CSSL IV, EASY5 etc., for overviews see Kreutzer (1986) and Kheir (1988)) follow the CSSL definition (Strauss, 1967). These tools solve problems of the type $dx/dt = f(t,x)$ if the user defines a Fortran-like procedure which calculates $f(t,x)$.

To allow a model to be used for different purposes, it should be declarative and not procedural. It should describe facts and relations (equations) and not be a calculation procedure. A declarative model is multipurpose, since it can be manipulated automatically to generate efficient code for simulation, code for calculation of stationary points, linear representations, descriptions which are accepted by other existing packages etc. Models of controllers can be used for automatic generation of the control software or to generate layouts for special purpose analog or digital VLSI circuits which implement the controller.

A declarative model is usually also closer to the model developer's perception of the physical reality, and therefore, development of new models is easier. When developing a model from first principles for a physical system one uses fundamental laws as mass balances, energy balances and phenomenological equations. Model development as well as documentation are facilitated if the user can enter these equations as they are without having to transform them into a computational procedure. The risk of introducing errors during manual transformation is reduced. The natural declarative form for continuous time models are Differential-Algebraic Equation (DAE) systems, $g(t,\dot{x},x) = 0$. An overview of important properties can be found in Mattsson (1989a).

The kernel can allow any logical and mathematical framework such as differential-algebraic equations or difference equations to describe behaviour, but a basic idea is that behaviour descriptions should be declarative and equation based.

Structured models

To understand large models and to be able to reuse parts of models, good structuring facilities must be supported. A powerful modularization concept supports model development by beating complexity as well as it allows reuse of parts and building of models by putting together existing components.

Block diagrams is a common structuring tool. A block represents a submodel. The connections between the blocks show cause-and-effect relationships between inputs and outputs of the submodel. A connection is unidirectional saying that the value of an output should be calculated from the input connected. It means that the model developer must deduce the computational causality to define what are inputs and outputs to a submodel.

When making a model library, it is very inconvenient to define which of the terminals of a submodel that are outputs, because what are inputs and outputs of a submodel is not only a property of the submodel itself, but also of how it is used. As motivated above a model should not be a computational procedure. It should not be a procedure which can calculate the outputs when the values of the inputs and the internal state are given. The model development and simulation tools must be able to handle interactions with unspecified computational causality (non-directional interaction). A connection should only define a relation saying that two terminals A and B are equal, not define a compute statement $A := B$ or $B := A$. Ideas like this have been developed in connection with special purpose simulators. An example is SPICE (Nagel, 1975) for electrical circuits where the basic building blocks are four poles.

In a simulation language like CSSL (Strauss, 1967) model decomposition is handled by macros, which require specification of the causality of the interaction. Another drawback with the macro concept is that the model structure is not preserved at compilation. The macros are expanded at compilation and at simulation the model has no structure. The names of the states and the variables of the submodels are replaced by names like `QQQQ1`, `QQQQ2` etc that are generated automatically resolve potential name conflicts. In the simulation language Simnon (Elmqvist, 1975) blocks are included in the language, but it is necessary to specify causality.

Bond graphs (Karnopp and Rosenberg, 1971) is another way of describing a model. It works well if the components are coupled via energy exchange only.

Our proposal for model structuring is object-oriented and introduces a small, basic, common set of concepts; A collection of basic objects like *models* and *terminals* with specified properties and operations. The proposal is described further in the next chapter.

Automatic consistency checking

It is important to make the use of library models safe and reliable. A model component is an encapsulated entity with well defined interfaces. This prevents to a large extent unintended abuse. It would be nice if the user could get automatic warnings when making improper connections when putting together a model. To allow automatic consistency checks, the model developer must "supply" redundant information. Our concepts allow a model developer to supply such information as described in next chapter. However, it is not our aim to force a user who, for example, is in an exploratory phase, to specify things that the computer itself can deduce from the context. A model developer is hopefully better motivated to supply redundant information when he has tested the model and is going to include it in a public model library. Such information can be seen as a part of the model documentation.

Customized user interfaces

We believe that various users could agree upon the objects proposed, but that they want to have customized user interfaces with various textual and graphical presentations. The proposal focuses on the basic objects and their properties and allows integration of different customized user interfaces.

The concepts proposed are basic and are mainly intended for researchers and modelling and simulation specialists. Other user categories can be supported by building new user interfaces and new layers of tools. Such tools can allow an architect or a chemical engineer to describe his building or chemical plant and the assumptions in his own language. The tools should then generate the desired model in an explicit form as outlined below. It means that the generated model is readable and can be modified by the user. Today's "high-level" tools of this kind are too rigid. They produce canned, black box models which cannot be modified. The user is in trouble if some component is missing, since it is very difficult or even impossible for him to add new components.

4. Model Structures

An important conclusion from computer science is that modules should be encapsulated with well-defined interfaces. The idea is to support abstraction by separating the internal details of a model from its interface. It means also that internal details can be changed without affecting the way the module is used as a component.

The *model* is the kernel's basic structuring unit. It is an abstraction of some dynamic behaviour. A model consists of three parts: terminals, parameters and realizations. The terminals are variables which constitute a well-defined interface to describe interaction with the environment. Parameters are interface variables defined by the model designer to allow the user to adapt the description of behaviour.

Realizations

A realization is a description of model behaviour. A model user can use a model without having to bother about how its behaviour is defined internally and the model designer can and must define its behaviour without any assumptions about the environment.

One reason for treating a realization as a separate part within the model is that we want to have multiple realizations. Different realizations can give more or less refined descriptions of the behaviour or they can define the behaviour for different working conditions or phases of a batch process. The user can choose the appropriate realization for each particular use.

We distinguish between primitive realizations and structured realizations. A structured realization is decomposed into submodels and its behaviour is described by the submodels and their interaction. The submodels can in turn have structured realizations which means that the model concept is hierarchical. A primitive realization is not decomposed into submodels, but its behaviour is described in some mathematical or logical framework as differential equations, difference equations etc.

Parameters

A parameter is a time invariant variable that can be set from outside to modify a realization. The burden of a user to set parameters can be relieved by letting the model developer provide default values. If a good default alternative is provided, the casual user could be left unaware about the flexibility and no extra burden is put on him. To support reparameterizations and alternative parameters, it is possible to define relations between parameters.

Terminals

Terminals can be viewed as variables which are shared by the internal description of the model and its environment.

It is natural to aggregate terminal variables, since the description of an interaction often involves several quantities. We propose two types of composite terminals: record and vector terminals. Their subterminals can be simple, record or vector terminals.

EXAMPLE 4.1—A pipe terminal
A terminal to describe the ends of a pipe or the inlets and outlets of pumps, valves and tanks can be defined as a record terminal

```
PipeTerminal IS A RecordTerminal WITH
  components:
    p IS A PressureTerminal;
    q IS A MassFlowTerminal;
    d IS A DiameterTerminal;
END;
```

having three components, which are simple terminals. The component d defines the diameter of the pipe or hole. □

Connections

Interactions between submodels of a structured realization are described by terminal connections. The term "connection" reflects what we are doing in the block diagram when describing an interaction. We will not discuss user interfaces here, but just point out that a block diagram is a good way of describing model structure. Elmqvist and Mattsson (1989) have developed a prototype simulator, where hierarchical block diagrams with information zooming are used to visualize the model structure. Information zooming means that the amount of information displayed in a block changes dynamically depending on its size on the screen.

A connection between two structured terminals means that their first components are connected to each other and so on recursively down to the level of simple terminals. There are two sorts of simple terminals: *across* and *through*. A connection between two across terminals means that they are equal. Examples of physical quantities are position, pressure, temperature and voltage. Through terminals have an associated direction (in or out) and connected terminals should sum to zero. Examples of through quantities are mass flow, energy flow, force, torque and current.

A simple terminal has an attribute defining the unit of measure with the SI unit as default. It is used for automatic introduction of proper scale factors in the connection equations, thus eliminating the need of user defined adapters.

It is important to note that generally the causality of a terminal (input or output) is not defined by the model designer but is inferred from the use of the model.

The semantics of a connection is kept simple, since we do not want to provide two different ways of describing complex behaviors. It is possible to describe complex interaction by introducing new submodels. It is also desirable to make the means to describe interactions independent of the frameworks used to describe the behaviour of primitive models.

EXAMPLE 4.2—Pipe terminals cont.
Assume that we want to model a system where a tank has a valve at the outlet. We then just connect the outlet terminal of the tank model to the inlet terminal of the valve model. The equations for the interaction saying that the pressures as well as the diameters should be equal and that the mass flows should sum to zero are deduced automatically from the connection. □

Consistency of connections

It is important to make the use of library models safe and reliable. The encapsulation of the models prevents to a large extent unintended abuse, but the terminals are dangerous holes in the wall. To allow automatic checks of connections, the model developer may add extra information, which also is useful for documentation.

Simple terminals have the attributes name of quantity and value range. The name of quantity is used used to check the consistency of connections. There is an international standard (ISO 31) for naming of quantities in different national languages like English or Swedish. Information about ranges of validity is used to test for unintended abuse during simulations.

A terminal component may be declared as time-invariant. Such a terminal is similar to a parameter. This has two complementary uses. First, a connection implies automatic propagation of parameter values from one submodel to another. Second, if the two connected parameter terminals have defined values, they must be equal for the connection to be consistent.

EXAMPLE 4.3—Pipe terminals cont.
Consider `PipeTerminal` in Example 4.1. The pressure component `p` can be defined by

```
PressureTerminal IS A SimpleTerminal
WITH
  attributes:
    value       := UNKNOWN;
    quantity    := pressure;
    unit        := kPa;
    direction   := across;
    variability := time_varying;
    causality   := UNKNOWN;
END;
```

The mass flow component `q` and the diameter component `d` are defined in analogous ways. An important difference is that mass flow is a through variable and the direction attribute should be set to `in` or `out`.

The variability of `d` ought to be set to `time_invariant` if the model does not allow the size of the pipe or hole to vary with time. It also allows automatic check of that two connected pipes are of the same diameter.

The terminal could also have a component indicating medium, which can be used for consistency checking or parameter propagation. For example, we can check that water pipes are connected to water pipes. □

Unspecified terminal attributes

To allow exploratory model development and prototyping, a declaration of a terminal may leave attributes unspecified as long as necessary information can be deduced from the context. Unspecified attributes make it possible to develop generic models. To support consistency checks of generic models, the model developer can specify relations between unspecified attributes. See Mattsson (1989b).

5. Object-Oriented Representation

In this chapter we will outline the conceptual design of a kernel for model representation. The basic entities, relations between entities and operations on them are discussed.

Object-oriented programming has been an increasingly popular methodology for software development. Increased programmer productivity, increased software quality and easier program maintenance are the objectives for this new methodology. Object-oriented programming supports these objectives by facilitating modularization and reuse of code. We will here show that ideas from object-oriented programming are useful also for model representation. For a brief introduction to object-oriented programming see Stefik and Bobrow (1986).

Basic model objects

Models and model components are *objects* in the kernel for model representation. An object has a unique identity within the system and it contains a collection of attributes. There is a number of important types of objects recognized in the kernel. They are representations of model structuring entities discussed in the previous section:

- models,
- terminals,
- parameters and
- realizations.

The last three object types can be used as components of models.

Class objects and relations

In our proposal, all model objects are represented as classes. In object-oriented programming a class describes the properties common to a set of similar objects – it defines an object *type*. For this reason, a model defines a component type rather than a particular instance of a component; the same applies to realizations, terminals, etc. A class can have a number of *attributes* which can be simple variables or relations to other model objects.

There are a three important relations which can be established between model objects. These are:

- has – part-of
- subclass – super class
- connection

The *has*-link is typically used between a model and its terminals, parameters and realizations. Further, a structured realization has this kind of relation to other models indicating the submodels. A has-link is stored as an attribute of the owner. The inverse relation is called *part-of.*

One class can be defined to be a *subclass* of another class – the super class. The subclass will inherit all properties of the super class in addition to the locally defined properties. *Inheritance* is an important concept in object-oriented programming and its use in this context will be discussed below.

A *connection* is a symmetric relation between two terminals and it is stored as an attribute of a structured realization.

EXAMPLE 5.1—Tank model
In this example we will show a model of a tank written in Omola (Object-Oriented Modelling Language) (Andersson, 1989a,b). Omola is a declarative language for model representation that has been designed to support our proposed concepts.

```
Tank IS A Model WITH
 terminals:
  inlet  IS A InPipeTerminal;
  outlet IS A OutPipeTerminal;
  level  IS A OutTerminal;
 parameters:
  area TYPE real := 1.0;
  roh  TYPE real := 1.0;
 realization:
  normalBehaviour IS A SetOfDAE WITH
   equations:
    area*dot(level) =
      inlet.q - outlet.q;
    inlet.p + level*roh*g =
      outlet.p - roh*v*abs(v)/2;
    outlet.q =
      pi*(outlet.d/2)^2*v*roh;
 END;
END;
```

This code represents a tank model with three terminals, two parameters and a realization component stored as attributes. The inlet and outlet terminals are both pipe terminals as in Example 1, but with directed flow components. For `inlet` positive flow is into the tank and for `outlet` positive flow is out from the tank. The realization has three equation attributes. The first equation is a mass balance and the other two are derived from Bernoulli's equation. In Figure 5.1 we can see some of the objects involved and their relations represented graphically. □

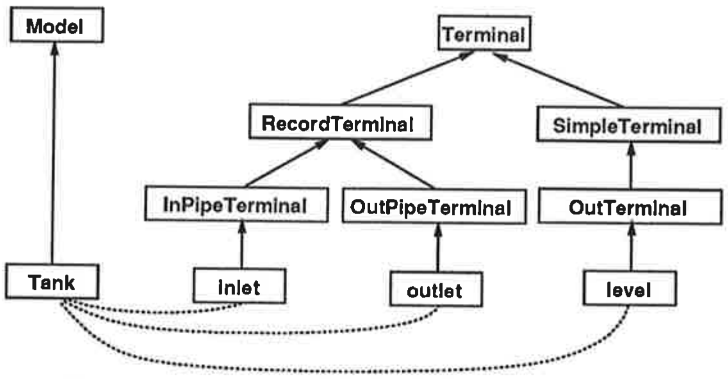


Figure 5.1 Some of the objects and their relations in the tank model. Subclass links are solid while has-links are dashed.

Inheritance

Inheritance is an intricate but powerful concept in object-oriented programming. When a class is defined to be a subclass of another class it will inherit all attributes and properties from the super class. The subclass is then free to add additional attributes or to redefine inherited attributes. Inheritance can be used to separate out some general attributes from a set of similar classes into a common super class.

Inheritance will facilitate reuse of models since carefully designed general models can be saved in libraries. These models or model components can be used as super classes of more specialized model objects. We have already seen how terminals have been defined in this way. The `inlet` and `outlet` terminals of the tank model are subclasses of `InPipeTerminal` and `OutPipeTerminal` which are specializations of the same super class `RecordTerminal`.

As an example of how models can be defined by specializations we can imagine a model of a regulator defining only the terminals: set-point, measure value and control value. This model can be specialized into different types of regulators by means of adding different realizations. We may then define a structured model like in Figure 5.2, containing the most general regulator model. The structured model can then be specialized to contain different regulator models.

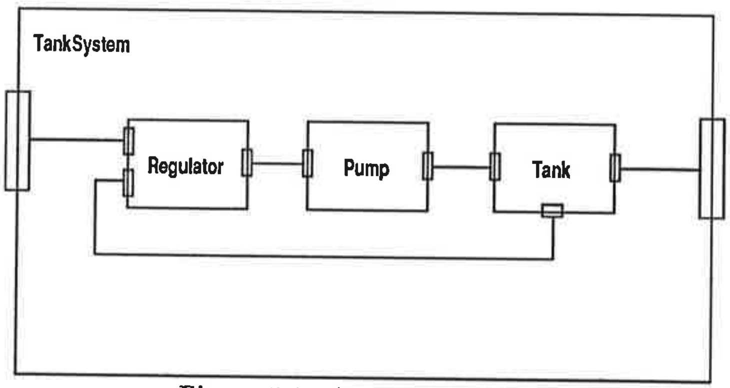


Figure 5.2 A structured model

Interpretation of model objects

Model structures represented in the kernel or in Omola code can be accessed and manipulated by different tools in a CACE environment. We may say that a particular tool that extracts relevant properties of a model *interprets* the model. Different tools may extract different properties and therefore, they interpret the model differently.

Since all model objects discussed so far are classes, i.e., they represent types rather than instances of model objects, one obvious interpretation is to use a model as a template to create a *model instance*. A model instance is, for example, needed when the model is going to be simulated. Then there must be representations for each particular model object and state variable. The instantiation procedure is recursive in the components and submodels. Typically when we want to simulate a model it is first instantiated then all equations are extracted from the primitive models and equations are generated from the connections. Second, the equations are sorted and turned into code that can be used by the DAE-solver. Since the model structure is maintained in the simulation model (the model instance) the user can access it the normal way, perhaps through its block diagram, and examine or change parameters and initial values.

As examples of other possible interpretations of model objects we can mention

- to generate a graphical picture of a system structure,
- to generate a text descriptions of a model for documentation,
- to generate a special purpose code, e.g., regulator code or
- to turn a model into a form accepted by a particular design package.

6. The User Interface

The user interface is a very important component in any computer based tool and in particular it is important in our proposed environment. A simple user interface has been implemented in our prototype in order to demonstrate the basic concepts. In this chapter we will first give a very brief overview of the current trends in design of interactive user interfaces. Then we will give a short description of the interface of the implemented prototype.

Current trends in user interface design

Human – computer interaction is currently a very active research area. Developments in computer hardware technology have made it possible to create very advanced user interfaces to application programs. However, the methods for designing such advanced interfaces are still rather primitive.

A current trend is to more and more separate the implementation of the user interface from the application program. For simple applications this can be done in a clean and natural way, but in many cases for more complicated application programs this is not a clean cut. Often a good user interface needs a substantial amount of "understanding" of the application. In other words, the user interface needs a model of the application. In this case we have the problem of keeping the application model consistent with changes in the application.

Another trend in interface design is to use higher level specifications of the user interaction. Commonly used are toolkits of various graphical objects and interactors, such as dialogue boxes, push buttons and menus of different kinds. They are often designed for a special computer or a special window manager. The InterViews (Linton et al., 1989) is an example of such a toolkit based on X Window System. A *User Interface Management System* (UIMS) contains a library of interactive objects like the toolkits but it also has a number of tools that helps the interface designer to put the objects together into a complete user interface. The interface designer may describe the interface on a more abstract level, sometimes by a declarative language. This means that the designer specifies what is to be done by the user interface rather than the exact details of how to do it. The designer may also use some formalism to describe dialogue such as transition graphs or BNF (push-down automaton). Some more advanced tools allow the interface designer to build the user interface in an interactive or semi interactive way with immediate feedback showing the current appearance of the interface. An introduction and survey of UIMS can be found in Mayers (1989).

The ongoing standardization of windowing systems and graphic input and output primitives makes it more attractive to develop and commercialize advanced UIMS software. In a few year's time such systems will probably be more commonly available at reasonable prices.

The prototype interface

In our prototype we have realized the importance of a reasonable good user interface. In the project we have not in particular studied user interface design as such, since the project have been focusing on representation of models rather than the presentation. However, in order to demonstrate the power and appropriateness of the underlaying representation a reasonably advanced interactive user interface had to be designed. We chose KEE[1] as the basic implementation tool for the prototype. One reason was that it provided some amount of support for building user interfaces. KEE uses object-oriented representation of graphical entities. Predefined primitive graphical objects can be specialized and combined into more advanced ones.

A graphical interface in our suggested modelling environment can not be clearly separated from the application — the model representation data base — because it is too much involved in the used data model. The approach taken instead, is to let the user interface operate directly on the model data base. The models represented in the data base may then contain additional information manipulated only by the user interface. For example, a model contains information about how it is presented on the screen, graphically or as text, menus of possible operations, etc.

Direct manipulation of models

The style of interaction in the prototype user interface is based on *direct manipulation*. Most objects, attributes and relations in the model data base can be represented on the screen. The screen representation can be a graphical icon, a diagram or a textual representation. In general every object is mouse sensitive and has an associated menu of operations.

The model data base in our prototype is divided into a set of *libraries*. A library is a collection of model objects and their attributes, and it can be saved and loaded from external memory. Objects in different libraries may have relations. The screen is separated into four important areas:

- an access window for loaded libraries,
- a library display window,
- an editor area and
- one or more general display windows.

The access window for loaded libraries displays a list of all loaded libraries where each entry is mouse sensitive and has an associated menu of library actions. The library display window shows the content of a selected library. For example, it may show the graphical icon of every model object in the library. Two important operations are implemented for most model objects; these are *display* and *edit*. These operations can of course be called from

[1] Knowledge Engineering Environment, KEE is a trademark of IntelliCorp, Inc.

the object's menu which is accessible through its icon but since they are very commonly used there is an alternative short cut. An object can be picked from the library display and an icon contour image can be dragged into an appropriate area of the screen. If an object is dragged into a display window the object will be displayed in that window. If an object is dragged into the editor area of the screen, the object can be edited.

In the editor area of the screen, one of a number of different editors may appear. The type of the object to edit determines which editor that will be invoked. There is a text editor for primitive realizations (equations) and other text definitions. A structure editor is invoked for block diagram editing of structured models and a form is invoked for editing the attributes of simple terminals.

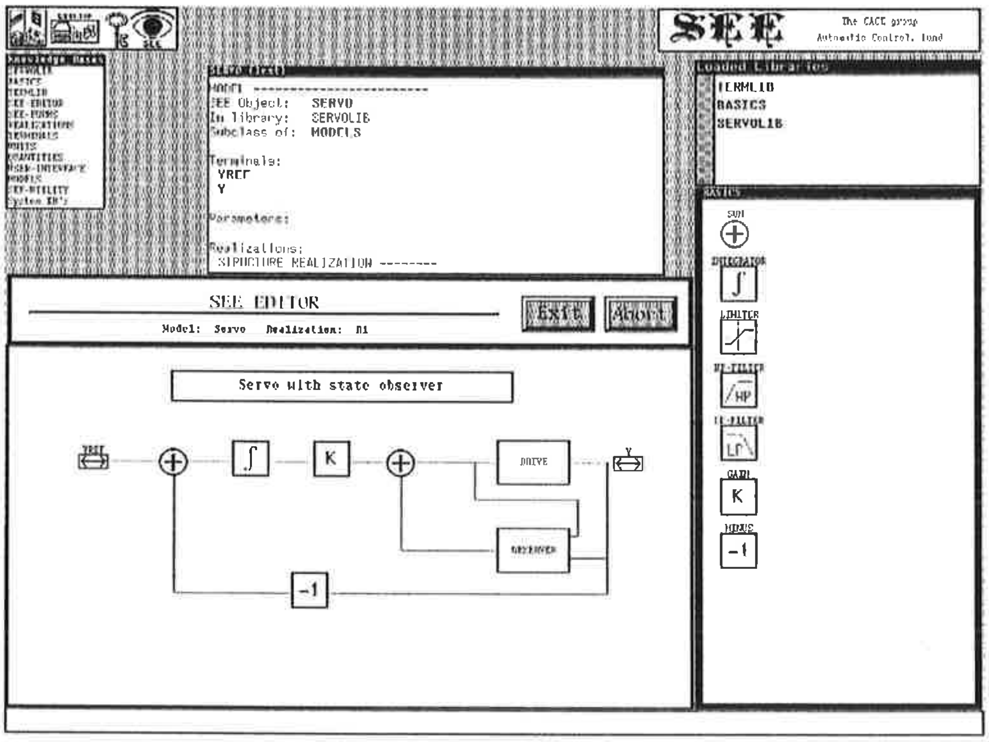

```
MODEL ------------------------
SEE Object:    SERVO
In library:    SERVOLIB
Subclass of:   MODELS

Terminals:
 YREF
 Y

Parameters:

Realizations:
 STRUCTURE REALIZATION --------
```

Figure 6.1 The prototype modelling environment.

7. An Application Study

The project has also included an application study which focused on modelling of chemical processes. The aim was to get some evaluation of the ideas and feedback from a real example. The application study has been performed by Bernt Nilsson, who is a chemical engineer interested in modelling. He has played the role of a user who wants to model a medium sized, typical chemical process plant that contains a reaction part with a tank reactor and two tubular reactors, and a separation part with three distillation columns in series.

The modelling work is described and discussed in Nilsson (1989), where also the model can be found. Since the application is a typical chemical plant, he presents an object-oriented modelling approach for chemical plants.

Chemical processes are often complex plants that are composed of a large number of components. However, chemical processes are often built as a number of subprocesses. In the application there are the reaction part and separation part. These subprocesses can be decomposed further into process components or unit operations. This decomposition is easily and neatly described by our hierarchical model decomposition concept. The process components are often standard process equipments such as pipes, pumps, valves, reactors, heat exchangers, distillation columns etc. that are used in different configurations in different processes. A model class allows reuse of a description in several instances and the inheritance mechanism allows adaptation of a model.

Nilsson (1989) describes ways of further decomposing chemical models. One interesting example is the medium and machine decomposition. It is of interest to separate the description of the process components from the descriptions of the chemical media. In today's simulation systems a model of for example a chemical reactor contains a reaction model which can only be modified by setting parameter values. A specification of a chemical reactor should contain the equations describing its thermodynamic and hydrodynamic properties, while the equations describing the reaction should be associated with the chemical media. Nilsson shows that the submodel concept allows a nice medium and machine decomposition.

Regular structures are common in chemical processes. For example, a distillation column may contain a few hundred trays connected in series. To handle this conveniently, Nilsson proposes matrices of submodels and a matrix notation to describe how they are connected. Finite element approaches to distributed parameter systems (partial differential equations) create also regular structures.

Parameterization and generic models are important to increase the flexibility and the reusability of models. The concepts proposed support Nilsson's basic needs of parameterizations. He states that it important to be able to parameterize structural properties like the number of chemical components flowing in a pipe. With matrices of submodels it is possible to let the number of trays in a distillation column be a parameter. He illustrates in several ways that inheritance allows powerful parameterizations. For example, it makes it simple to change the model of the medium in the distillation column.

Nilsson concludes that the proposed model representation is superior to existing ones, but it requires also a good model/user interface and number of tools to make a good modelling environment.

8. Conclusions

First, the results of the project is summed up. Then technology transfer is considered. Third, some more general experiences of software techniques and tools are discussed.

8.1 Results

Most of today's languages for continuous simulation follow the CSSL definition (Strauss, 1967). It has served well for over 20 years. We think it is time to capitalize on the enormous development of information technology and reconsider the foundations of model representation. Our proposal is a modest effort in that direction.

The major contribution of the project is that experience in model structuring, progress in numerical analysis and new ideas in object-oriented design are collected and turned into a coherent scheme for model representation. Our proposed model representation scheme is general, powerful, clean and easy to understand. The result is presented as a design proposal of a kernel for model representation. The kernel is intended as central model representation data base in an environment of tools for system engineering. The basic features of the kernel are:

- Declarative and equation based behaviour descriptions to make the models versatile and useful for various applications.
- Hierarchical models with well defined interfaces based on terminals and parameters.
- Terminal attributes for automatic check of connection consistency.
- A model may have several behaviour descriptions to support model versions and alternative behaviour.
- Object-oriented representation where classes with inheritance facilitates reuse and incremental model development.
- An internal representation which preserves the structure of models.
- The kernel allows integration of customized user interfaces and various tools.

A prototype implementation has been written in Common Lisp and KEE. To get some feedback and evaluation of the ideas, the project has also comprised an application study focusing on modelling of chemical processes. Experiences from the prototype and from the application study indicate that the ideas are sound and that the kernel proposal may indeed serve as a basis for a new generation of modelling, design and simulation tools.

The proposed kernel ought to be of interest in all areas of engineering and for all who use models and simulation. The ideas have been presented at conferences and the prototype has been demonstrated for a number visitors from industry and universities. People from many different areas of engineering who have struggled with similar problems of model representation, have found our solutions very interesting.

8.2 Development and technology transfer

The scientific results of the project are and will be made public through articles in international magazines and as conference contributions (see Appendix A). Some of the results have and will be published as licentiate and doctoral theses.

The contact net with Swedish and foreign universities and companies that develop CACE tools is extensive and functioning. We are now extending it to include also researchers and developers working with model development tools and simulation in general. The international conferences give good opportunities to exchange ideas and information and to make acquaintance with new people. Besides control engineering conferences we have also participated in conferences aimed at modelling and simulation in general as well as conferences aimed at special applications as chemical engineering and building simulation.

Implementation of the kernel

A very good way of transfering results like that of our project is of course to make the tools available to many people. The experiences from Simnon show that useful program components are a very good way of spreading new ideas and methods. Our prototype is written in Common Lisp and KEE. The advantage for us of using KEE was that the prototype could be implemented with a modest effort. However, since KEE is very expensive, we do not expect the prototype to be widespread.

To make our tools generally available, it is necessary to implement them using cheaper and more commonly available languages and software components. We think that it is not the task of a university to develop, market and maintain commercial and professional software. But we realize that we have a responsibility of transfer the results of our project and making them generally available. A project supported by STU (STU project 89-01837 "A kernel for modelling and simulation") has just been started to implement a kernel for model development and simulation, which someone else can develop further into a commercial product of the prototype kernel. C++ is used as the basic implementation language. An economic reality is that it is expensive to develop professional software and the market for CACE-products is relatively small. However, our kernel may be of interest in most areas of engineering and ought to have a much larger market. There are companies and groups that have expressed interest in making a commercial product. However, it is too early to make any predictions now and we welcome all proposals.

Application projects

Another good way of spreading new ideas and methods is to have joint application projects with developers and users. However, to be able to transfer the results in application projects, implementation of the tools are needed. In application projects the developers get feedback and can modify and improve the tools. For a special application they can develop customized tools and user interfaces. Model libraries can be built which can be of use not only for the participating part, but also for a whole line of business.

We are planning to run a number of application projects. However, as pointed out above, we need implemented tools and the implementation project has therefore been given priority. A proposal for an application project together with Sockerbolaget has been submitted to STU's research program

DUP. The application is modelling of sugar crystallization. Simulation and simulators have a central role in DUP, which aim is to investigate how process operators' tasks can be supported by computer based tools. It is natural to set up and finance application projects in DUP. STU's program "Applications of the information technology" is another possible source of financing application projects.

Standardization

Much could be gained if we could agree upon a common set of ideas. It is time to lay the foundation for a new standard for model representation. IFAC has a working group on standards for CACSD Software. We are participating in this work. It has not addressed non-linear systems yet, but it has focused on linear systems.

It may be remarked that to build flexible model libraries we must also agree on common principles for model development. This is a hard task, but it might be possible to achieve in certain application areas.

There is an international association IBPSA (the International Building Performance Association), which promotes the science of building performance simulation in order to improve the design, construction, operation and maintenance of all types of buildings. IBPSA's international membership includes architects, engineers, building managers, academics, software developers, and government representatives concerned with building performance. IBPSA organized a conference Building Simulation '89 on June 23–24, 1989 in Vancouver, Canada. At this conference Per Sahlin, The Swedish Institute of Applied Mathematics, ITM, Stockholm and Edward Sowell, California State University, Fullerton, California presented a proposal for a neutral format for building simulation models to allow users to share models (Sahlin and Sowell, 1989). This proposal is inspired and influenced by the results of our project.

8.3 Experiences of software techniques and tools

The project has dealt with design of tools for model development and simulation and to do this we have exploited ideas, approaches, methods and techniques from computer science as well as used existing software. Hence we may also ask what we have learned that can be of more general interest.

Object oriented programming

Object oriented programming (for overview see e.g. Stefik and Bobrow, 1986) is a technique for structuring programs and to support reuse. The basic ideas are data abstraction and inheritance.

Objects and abstraction is natural in engineering. Block diagrams and and other kinds of schematics and flow graphs are common. Blocks have often well defined interfaces. In control engineering it is common to talk about input/output models, where only the relations between the inputs and outputs are known. Nothing is then said about the internal structure or the implementation of it. When the model also defines internal structure, we speak about internal models.

Although the ideas of object oriented programming are or at least seem to be natural, it is not self evident how to use them in a special application. Zobel and Cummings (1989) discuss use of object orientation for digital signal applications. They had found that it is many cases not obvious whether operations should be implemented as methods or as special processing objects. For example, should FFT be a method of signals or a special object (machine)? They were going to carry through both approaches to get a deeper insight and experiences.

In discrete event simulation the models can perform the simulation themselves by sending messages to each other. This is not possible in continuous time. The differential-algebraic equations must be solved simultaneously. It could be done in an object oriented fashion by having a Solver object that collects the equations from all the models, solves them and returns the result to the models.

We have used object orientation on several levels. First, for the architecture of the system to get an flexible and extendible integrated environment which allows customized user interfaces. Second, the modelling concepts are object oriented. Third, the internal model representation is also object oriented.

A kernel for model development must allow interactive definition and creation of new model classes (types). Interactive languages like KEE, CLOS, Smalltalk allow interactive definition of new classes and a model class can basically be implemented as a class in the implementation language. In compiled languages like Simula and C++, it is not possible to define new classes interactively. It means that it is not possible to represent model classes directly as classes in the implementation language. An extra layer to handle definition of new model classes and inheritance between model classes must be implemented.

Databases

Databases are central. We need them to store models, parameter data, measurements, results of calculations etc. Common representations are needed to make the tools integrated.

Today's databases can handle a large set of independent data efficiently, but in CACE the amount of data are moderate, but the relations are complex. For example, a model may be a linear version of another model at a certain operating point for some given parameter values. Object oriented databases is a promising approach.

Graphics and user interfaces

Computer graphics gives good possibilities to improve the user interface. It can be used to make concepts, properties, structures and other information more concrete. Direct manipulation is an interesting technique which allows the user to operate on visual objects and get immediate visual feedback. Visual metaphors must be selected carefully to give the user a correct conception.

Graphics must be designed carefully to be useful and endurable in the daily use. The primary use of graphics should not be spectacular demonstrations.

Unfortunately, it is laborious to implement graphics. First, portability is a major concern. As a user you want to have a homogeneous environment. The advantages of having a standard window system for all CACE programs are quite obvious and uncontroversial, but it should be noted that CACE programs are not the only use of a workstation. The user will use the native, vendor-supplied window system, and would therefore prefer that one also in

CACE programs. The same also applies for text editors. The situation seems to improve and X Window System is today a de facto standard. Second, there are today very few tools available for definition and implementation of user interfaces, but it is an active area. Hopefully there will be commercial user interface management systems (UIMS) available within a few years.

AI and expert system techniques

The complexity of AI has speeded up and influenced the development of powerful workstations, high interactivity, computer graphics, animation, object orientation, direct manipulation etc.

We consider the expert system technique as a useful and powerful programming technique. The kernel does not itself contain any expert system, but we have exploited ideas on information representation and declarative programming; equations to describe behaviour. Rules can be used to define events. Deduction of unspecified model attributes and consistency checking can be implemented by rule based systems.

Some people claim that they have knowledge based simulation when they provide simple model libraries. Knowledge based simulation is, however, in our opinion more than providing a model library. There should be facilities that assist the user to select the proper models and model versions as well as to evaluate the results.

Symbolic manipulation and computer algebra

The increasing computing capacity makes it possible to perform symbolic manipulation. The user can give his problem on for him a suitable form. Symbolic manipulation can then be used to simplify the problem and to generate descriptions that the numerical tools need. Analytic expressions may give better insight than tables of numerical values.

Existing commercial packages for computer algebra such as MACSYMA, REDUCE, Scratchpad, Maple and Mathematica are powerful. Unfortunately, it is not easy to use exisiting packages for computer algebra in other tools. They are interactive and assume that they are run by human beings. The results returned from the packages are on a format intended for human beings. They are not built to be run or called by other programs. They can of course be run as separate processes and comunication can be done via pipes, mailboxes etc. depending on the operating system. The difficulty is to decode the text strings returned by the package. The reference manuals do not give any formal specification of the format. So if we want to write a program to decode it, we have first to investigate what is returned. All this is laborious to do but the situation is even worse. Since the format is not formally specified, a new release of the package may change (improve) the format.

Libraries of routines for symbolic manipulation, like the numerics libraries would be very useful.

Acknowledgements

The authors would like to thank Professor Karl Johan Åström, Bernt Nilsson and Dag Brück for many useful discussions and Tomas Schönthal for his work on the prototype. This work has been supported by the National Swedish Board for Technical Development (STU) under contracts 87-2503 and 87-2425.

References

ANDERSSON, M. (1989a): "An Object-Oriented Modelling Environment," *Proceedings of the 1989 European Simulation Multiconference*, Rome, June 7–9, 1989, pp. 77–82.

ANDERSSON, M. (1989b): "Omola – An Object-Oriented Modelling Language," Report TFRT-7417, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ANON (1987): "Challenges to Control: A Collective View," *IEEE Transactions on Automatic Control*, **AC-32**, No. 4, April 1987, 275–285.

ELMQVIST, H. (1975): "Simnon – User's Manual TFRT-3091, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden,".

ELMQVIST, H. and S.E. MATTSSON (1989): "Simulator for Dynamical Systems Using Graphics and Equations for Modeling," *IEEE Control Systems Magazine*, **9**, 1, 53–58.

FLEMING, W.H. (Ed.) (1988): *Future Directions in Control Theory – A Mathematical Perspective*, SIAM Reports on Issues in the Mathematical Sciences, Society for Industrial and Applied Mathematics, SIAM, Philadelphia.

KARNOPP, D. and ROSENBERG, R. (1971): *System Dynamics — A unified approach*, Wiley, New York.

KHEIR, N.A. (Ed.) (1988): *Systems Modeling and Computer Simulation*, Marcel Dekker, Inc., New York.

KREUTZER, W. (1986): *System Simulation – Programming styles and languages*, International Computer Science Series, Addison-Wesley.

LINTON, M.A., J.M. VLISSIDES and P.R. CALDER (1989): "Composing User Interfaces with InterViews," *IEEE Computer*, **22**, 2, February 1989.

MATTSSON, S.E. (Ed.) (1987): "Programplan för ramprogrammet Datorbaserade hjälpmedel för utveckling av styrsystem (Computer Aided Control Engineering, CACE)," Final Report TFRT-3193, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. (1989a): "On Modelling and Differential/Algebraic Systems," *Simulation*, **52**, No. 1, 24–32.

MATTSSON, S.E. (1989b): "Modelling of Interactions between Submodels," *Proceedings of the 1989 European Simulation Multiconference*, Rome, June 7–9, 1989, pp. 63–68.

Mayers, B. (1989): "User-Interface Tools: Introduction and Survey," *IEEE Software*, **6**, 1, January 1989, 15–23.

Nagel, L.W. (1975): "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Technical Report ERL-M520, Electronics Research Lab, University of California Berkeley.

Nilsson, B. (1989): "Structured Chemical Process Modelling — An object oriented approach," Lic Tech thesis TFRT-3203, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Sahlin, P. and E.F. Sowell (1989): "A Neutral Format for Building Simulation Models," *Proceedings of Building Simulation '89*, Vancouver, June 23–24, 1989, The International Building Performance Simulation Association, pp. 175–180.

Stefik, M. and D.G. Bobrow (1986): "Object-Oriented Programming: Themes and variations," *AI Magazine*, **6**:4, 40–62.

Strauss, J.C. (Ed.) (1967): "The SCi Continuous System Simulation Language (CSSL)," *Simulation*, Dec 1967, 281–303.

Zobel, R.N. and A.M. Cummings (1989): "Developments in an Object Oriented Environment for DSP Applications," *Proceedings of the 1989 European Simulation Multiconference*, Rome, June 7–9, 1989, pp. 263–268.

A. Published Papers and Conference Contributions

ANDERSSON, M. (1989): "An Object-Oriented Modelling Environment," in G. Iazeolla, A. Lehman, H.J. van den Herik (Eds.): *Simulation Methodologies, Languages and Architectures and AI and Graphics for Simulation*, 1989 European Simulation Multiconference, Rome, June 7–9, 1989, The Society for Computer Simulation International, pp. 77–82.

ANDERSSON, M. (1989): "An Object-Oriented Language for Model Representation," IEEE CACSD'89, Tampa, Florida, December 16, 1989.

BRÜCK, D.M. (1988): "Modelling of Control Systems with C++ and PHIGS," *Proceedings of the USENIX C++ Technical Conference*, Denver, Colorado, October 17–20, 1988, pp. 183–192, Also available as report TFRT-7400, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

BRÜCK, DAG M. (1989): "Experiences of Object-Oriented Development in C++ and InterViews," *Proc. TOOLS'89*, Paris, France, November 13–15, 1989, Also available as report TFRT-7418, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQVIST, H. and S.E. MATTSSON (1989): "Simulator for Dynamical Systems Using Graphics and Equations for Modelling," *IEEE Control Systems Magazine*, **9**, 1, 53–58.

MATTSSON, S.E. (1988): "On Model Structuring Concepts," *Preprints of the 4th IFAC Symposium on Computer-Aided Design in Control Systems (CADCS)*, August 23–25 1988, P.R. China, pp. 269–274.

MATTSSON, S.E. (1989): "On Modelling and Differential/Algebraic Systems," *Simulation*, **52**, No. 1, 24–32.

MATTSSON, S.E. (1989): "Modelling of Interactions between Submodels," in G. Iazeolla, A. Lehman, H.J. van den Herik (Eds.): *Simulation Methodologies, Languages and Architectures and AI and Graphics for Simulation*, 1989 European Simulation Multiconference, Rome, June 7–9, 1989, The Society for Computer Simulation International, pp. 63–68.

MATTSSON, S.E. (1989): "Concepts Supporting Reuse of Models," *Proceedings of Building Simulation '89*, Vancouver, June 23–24, 1989, The International Building Performance Simulation Association, pp. 175–180.

NILSSON, B. (1989): "Structured Modelling of Chemical Processes with Control Systems," Annual AIChE Meeting 1989, San Francisco, Nov 5–10, 1989.

NILSSON, B., S.E. MATTSSON and M. ANDERSSON (1989): "Tools for Model Development and Simulation," in Larsson, J.E (Ed.): *Proceedings of the SAIS '89 Workshop*, AILU—The AI Group in Lund, Lund University and Lund Institute of Technology.

B. Reports

Lic Tech Thesis

Nilsson, B. (1989): "Structured Chemical Process Modelling — An object oriented approach," Lic Tech thesis TFRT-3203, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Master Theses

Granbom, E. and T. Olsson (1987): "VISIDYN – Ett program för interaktiv analys av reglersystem," Master thesis TFRT-5375, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Johansson, M. (1988): "Interaktiv plottning av mätdata i flera dimensioner," (Interactive plotting of measurement data in several dimensions), Master thesis TFRT-5390, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Jeppsson, U. (1988): "An Evaluation of a PHIGS Implementation for Full Graphics Control Systems," Master thesis TFRT-5389, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Nilsson, A. (1988): "Object-oriented Graphics for the Future Instrument Panel," Master thesis TFRT-5382, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Vallinder, P.A. (1988): "Some Methods for Tearing of Differential/Algebraic Systems," Master thesis TFRT-5384, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Other reports

Andersson, M. (1989): "Omola – An Object-Oriented Modelling Language," Report TFRT-7417, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Brück, D.M. (1989): "Scones — An Interactive Block Diagram Editor for Simnon," Report TFRT-7423, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Mattsson, S.E. and M. Andersson (1989): "A Kernel for System Representation," Report TFRT-7429, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Mattsson, S.E. and K.J. Åström (1988): "The CACE Project— Steering Committee Meeting, 1987-11-25," Report TFRT-7375, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Mattsson, S.E. (1988): "The CACE Project— Steering Committee Meeting, 1988-06-01," Report TFRT-7395, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. (1989): "The CACE Project— Steering Committee Meeting, 1988-11-23," Report TFRT-7412, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E. (1989): "The CACE Project— Steering Committee Meeting, 1989-09-08," To appear, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NILSSON, B. (1987): "Experiences of Describing a Distillation Column in Some Modelling Languages," Report TFRT-7362, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

C. Lectures

Oct 2, 1987. Sven Erik Mattsson: "Hibliz," Studsvik, Nynäshamn, Sweden.

Feb 29, 1988. Sven Erik Mattsson: "Overview of the CACE project and Hibliz," Foxboro Company, Foxboro, Massachusetts, USA.

March 16, 1988. Sven Erik Mattsson: "Hibliz and the expert system interface for Idpac," Department of Information Processing, Umeå University, Umeå, Sweden.

March 16, 1988. Sven Erik Mattsson: "The CACE project," the Department of Information Processing, Umeå University, Umeå, Sweden

April 26, 1988. Sven Erik Mattsson: "The CACE project," Intelligent Automation Laboratory, Department of Electrical and Electronics Engineering, Heriot-Watt University, Edinburgh, Scotland.

May 20, 1988. Sven Erik Mattsson and Bernt Nilsson: "The CACE project and tools for model development and simulation," the management group of the DUP project at STU, Stockholm, Sweden.

Aug 23, 1988. Sven Erik Mattsson: "On Model Structuring Concepts," the 4th IFAC Symposium on Computer-Aided Design in Control Systems, CADCS'88, August 23 – 25, 1988, Beijing, P.R. China.

Sept 22, 1988. Sven Erik Mattsson: "Methods and languages for development of simulation models," workshop on the use of simulators in the process industry, arranged by the STU-program DUP, Stockholm, Sweden

Oct 20, 1988. Dag Brück: "Modelling of Control Systems with C++ and PHIGS," the USENIX C++ Technical Conference, October 17 – 20, 1988, Denver, Colorado, USA.

Nov 2, 1988. Karl Johan Åström: "Drum water modelling," a one day seminar on Simulation and Advanced Control of Power Plants at Sydkraft, Malmö, Sweden

Nov 2, 1988. Sven Erik Mattsson: "Future Modelling and Simulation Environment," a one day seminar on Simulation and Advanced Control of Power Plants at Sydkraft, Malmö, Sweden.

Nov 11, 1988. Dag Brück: "Object oriented design in C++," Ericsson Radar Systems, Mölndal, Sweden.

Dec 14, 1989. Sven Erik Mattsson: "Future Modelling and Simulation Environment," Workshop on Future Research Needs in CACSD, Cambridge, 14 & 15 December 1988. Arranged by the Science and Engineering Research Council, SERC in UK.

March 14, 1989. Dag Brück "Experiences of C++ and UNIX," STFI, Stockholm, Sweden

May 7, 1989. Bernt Nilsson: "Tools for Model Development and Simulation," SAIS '89, the annual workshop of the Swedish Artificial Intelligence Society, Lund, Sweden

May 31, 1989. Sven Erik Mattsson: "The CACE project and new tools for model development and simulation," the council of SIGSIM, Sweden.

June 8, 1989. Mats Andersson: "An object-oriented modelling environment," ESM'89, the 1989 European Simulation Multiconference, June 7–9, 1989, Rome, Italy.

June 8, 1989. Sven Erik Mattsson: "Modelling of interaction between submodels," ESM'89, the 1989 European Simulation Multiconference, June 7–9, 1989, Rome, Italy.

June 23, 1989. Sven Erik Mattsson: "Concepts supporting reuse of models," Building Simulation '89, Vancouver, Canada.

Särtryck ur IEEE Control Systems Magazine, Jan 1989

Simulator for Dynamical Systems Using Graphics and Equations for Modeling

Hilding Elmqvist and Sven Erik Mattsson

ABSTRACT: New workstations with high-performance graphics offer new possibilities for man-machine interaction. This paper presents a prototype simulator for dynamical systems, called Hibliz (HIerarchical BLock diagrams with Information Zooming), which explores some features of modern computer graphics. Hibliz supports hierarchical block diagrams to describe the model decomposition and interconnection structure. The user can scroll, pan, and zoom the block diagram continuously in real time. Zooming controls the amount of information displayed. When zooming in on a block, it changes from an annotated box to a representation showing internal structure with increasing detail. Since the block diagrams can be hierarchical, it is possible to make the description at each level simple and clear. Hibliz also simplifies model development by allowing submodels in the form of ordinary differential and algebraic equations rather than assignment statements for derivatives and algebraic variables.

Introduction

It is difficult for a human to develop and handle models of large and complex systems, because most humans are unable to deal with many entities simultaneously. Consequently, a system for model development and simulation ought to have structuring facilities so that the user can view a model from different viewpoints, each having only a small number of entities. Up to now, it has been difficult to design and implement such facilities, because structural properties are not easy to represent textually. The decomposition of a model into submodels with interconnections is more easily described graphically. The current trends in scientific personal computers will make graphical displays commonly available. This will revolutionize the man-machine interaction. Apple's Macintosh is a clear indication of what we could expect. Such machines will offer great opportunities to invent more efficient problem-solving tools.

This paper presents a prototype system for model development and simulation. The simulator explores some of the possibilities of new workstations with high-performance, real-time graphics. Special attention has been given to the use of graphics to describe structural properties. Hierarchical block diagrams are used to describe the model decomposition and the interconnection structure. By moving a mouse and pressing its buttons, the user can scroll, pan, and zoom the block diagram continuously in real time. Information zooming is used to control the amount of information displayed. When zooming in on a block, it changes from an annotated box to a representation showing the internal structure with increasing detail. The concept of information zooming was introduced by Elmqvist [1]. The prototype simulator is called Hibliz (HIerarchical BLock diagrams with Information Zooming).

Hibliz is a simulator for dynamical systems, described by sets of ordinary differential equations and algebraic equations. The structuring concepts proposed also can be used for more general systems with both continuous-time and discrete-time submodels; however, the algorithm presently included for simulation can handle only continuous-time models. Hibliz allows the submodels to be described in the form of equations instead of assignment statements to facilitate modeling and use of model libraries.

The model description concepts of Hibliz are discussed in the following section. The operation of Hibliz is described in the next section and then implementation is explained.

Model Description Concepts

Two basic principles can be used to structure a model: abstraction and modularization. The essence of abstraction is to extract important properties while omitting insignificant details. Different levels of abstraction are defined, allowing the system to be viewed with increasing detail. The first abstraction level for a model might be just its name or icon. The next level might describe usage and external behavior, and a third level might detail its internal behavior. The amount of information increases at lower abstraction levels. Modularization can be used to maintain useful views with a limited number of related concepts. Modularization means that the information at a certain abstraction level is decomposed into smaller entities.

The concept of this paper is that a graphical description of the structure is easier to understand than a textual description. Modularization is achieved by use of block diagrams. To support abstraction, information zooming and hierarchical block diagrams are proposed. Multiple windows are used to support further multiple views of the model.

The use of hierarchical block diagrams and information zooming will be illustrated by an example. Please remember that when sitting at the terminal one can scroll, pan, and zoom in the windows, but that the dynamical aspects are lost in a paper that can show only snapshots of the screen.

An Example

As an example, consider a model of a thermal power plant. The block diagram in Fig. 1 shows the major components and their interaction. The annotated boxes represent submodels, and the lines between the boxes indicate interaction between the submodels. To the left in Fig. 1, we find the model for the combustion chamber. It delivers energy to the boiler, heaters, and reheaters in the turbine part. The boiler produces steam, which is heated in the superheaters. The steam then goes to the turbines via the steam valve. From the turbine system, the steam enters the feed water system. Extract steam from the turbines is used to preheat the feed water. The feed water goes to the boiler, and to sprayers in the heater as well. The equations describing the system are typically mass and energy balances. External functions for interpolating in steam tables are also required. The model has 470 variables; in its textual form, it is 1200 lines long, including layout information.

By pressing the right mouse button when moving the mouse, one can scroll and pan

Presented at the IEEE Control Systems Society Third Symposium on Computer-Aided Control System Design, Arlington, Virginia, September 24–26, 1986. Hilding Elmqvist is with SattControl AB, Development Center, P.O. Box 62, S-221 00 Lund, Sweden. Sven Erik Mattsson is with the Department of Automatic Control, Lund Institute of Technology, Box 118, S-221 00 Lund, Sweden.

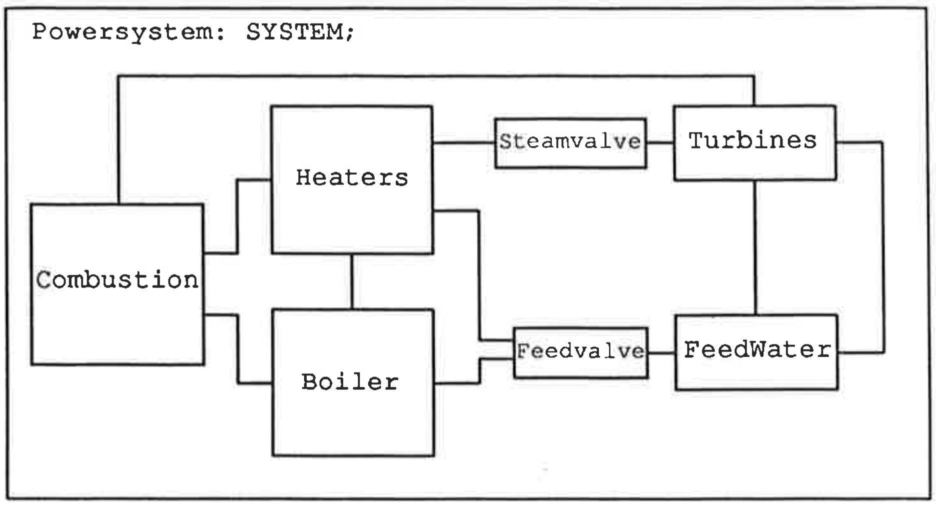


Fig. 1. Power system model.

The user can point at an object in any window and ask for automatic scrolling, panning, and zooming to this object in the interaction window.

the picture continuously. If both the right and middle buttons are pressed, one zooms. When one zooms in on the diagram, the blocks open up and show their internal details. Zooming further (Fig. 2), the internal description of the boiler can be seen. It is a new block diagram.

Model Decomposition

Hibliz supports the use of hierarchical block diagrams as a tool to handle complexity. A basic rule is that a block diagram should be simple and contain only a small number of blocks. The selection of module boundaries is guided by one's perception of the problem space. If a first attempt at structuring results in too many blocks, it is advisable to introduce a new hierarchy. There is almost always interaction (coupling) between modules. In order to be useful, a decomposition must be chosen in such a way that the external interaction complexity is small compared with the internal complexity. There should not be crisscross lines between the blocks. Furthermore, the entities in a module should be related (cohesion).

Modularization gives many advantages. It simplifies the modeling. It makes the model more flexible and easier to adapt and manage. One can also build and use libraries of models. Technical systems are often built in a modular way and composed of standard components. The behavior of these standard components may be well known, and good, generally accepted models already may exist.

Multiple Windows

The user can create new windows for viewing a model. One of these windows is the current interaction window for scrolling, panning, and zooming. To help the user keep track, rectangles outline the parts of a window that are also shown in other windows.

Interaction Structure

Now consider the interaction between the models. In the first level, the block diagram (Fig. 1) shows which models interact. The next level (Fig. 2) is concerned with how models interact, i.e., which variables are involved. The graphical representation for a submodel consists of two rectangles; one inside the other. The descriptions of interfaces are placed at the border of the submodel between the inner and outer rectangles. At the most detailed level, the effect of interaction can be seen in the equations containing interaction variables.

A model is an encapsulated entity, and the interaction variables are the only variables visible from the outside. The interaction variables are associated with submodels. They cannot be associated with connections, because it should be possible to develop a submodel without knowing in what environment it will be used. This is necessary to allow for model types and model libraries. A model often interacts with several other models, implying that the formal interaction variables should be grouped corresponding to the different possible connections. Such groups are called *interfaces*. Interfaces may have a hierarchical structure.

For example, consider the model Boiler in Fig. 2. To the right, the interface Iwater of Boiler can be seen. It models the incoming feed water to the Boiler. This interface is connected to the right interface of Economizer to model that the feed water goes to the Economizer. The feed water is heated in the Economizer. It then leaves the Economizer and flows into the Drum. This is modeled by the connection between the left interface of Economizer and the right interface of Drum. In Fig. 3, we can see that the interfaces of Economizer have three components: flow rate, enthalpy, and steam pressure.

A connection between two structured interfaces means that their corresponding components are connected. The number of components must be the same in the two interfaces. Primitive interface components also may be used to pass through a structured connection to submodels. The user needs only to specify the interfaces of nonprimitive models to the degree of detail necessary to draw the block diagram within the actual block. The connection between the interface Iwater of Boiler and the right interface of Economizer illustrates this.

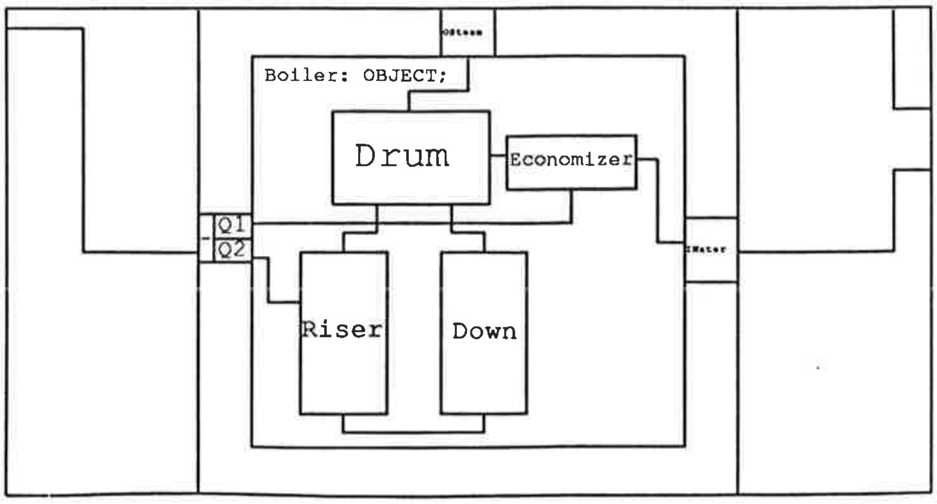
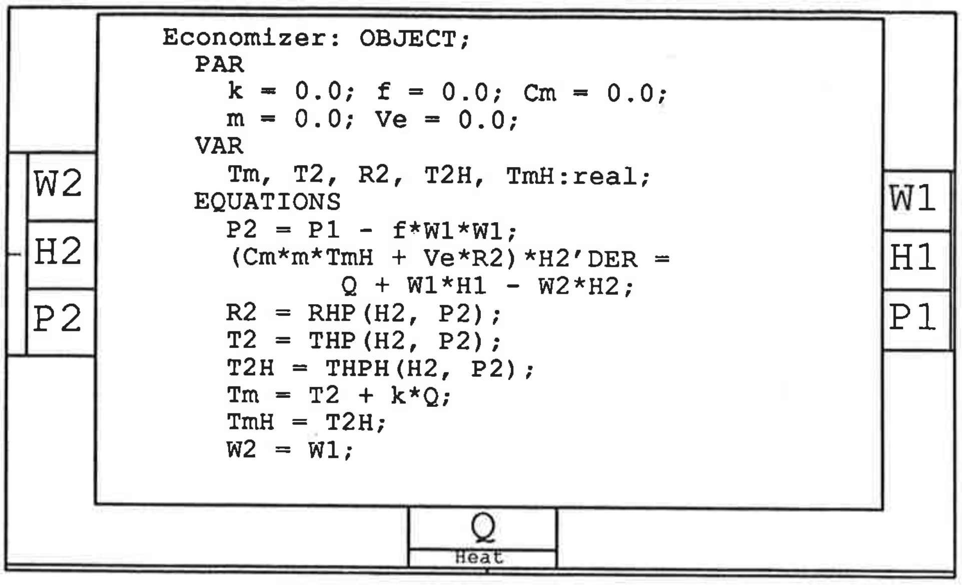


Fig. 2. Power system zoomed in at Boiler.



```
Economizer: OBJECT;
  PAR
    k = 0.0; f = 0.0; Cm = 0.0;
    m = 0.0; Ve = 0.0;
  VAR
    Tm, T2, R2, T2H, TmH:real;
  EQUATIONS
    P2 = P1 - f*W1*W1;
    (Cm*m*TmH + Ve*R2)*H2'DER =
            Q + W1*H1 - W2*H2;
    R2 = RHP(H2, P2);
    T2 = THP(H2, P2);
    T2H = THPH(H2, P2);
    Tm = T2 + k*Q;
    TmH = T2H;
    W2 = W1;
```

Fig. 3. Economizer model.

Behavior

Before discussing the semantics of connected variables, consider a primitive submodel. The model Economizer is one example. As before, zoom manually, but there is also a facility for quick automatic zooming. Pointing with the cursor at the outer rectangle of a model and pressing the middle button, Hibliz automatically zooms in on the model (Fig. 3).

The behavior of models at the lowest hierarchical level is described by equations in textual form. An equation should have the form

$$\text{expression} = \text{expression}$$

We will not describe the syntax and semantics of expressions in full detail (see [1]). Expressions have the usual syntax with arithmetic, relational and Boolean operators. Conditional parts (if-then-else expressions) as in, for example, Algol 60 are also allowed. Common mathematical functions such as sin, cos, and exp are available. The language supports simple integer, real, and Boolean variables. Hibliz also provides a mechanism for incorporating additional functions written in Pascal. These functions can be used directly in the model. For example, RHP, THP, and THPH used in the model Economizer (see Fig. 3) are such external functions implementing steam tables.

There are four kinds of variables: constants, parameters, interface variables, and internal variables. Constants and parameters are considered to be constant during a simulation, but the user can change the values of parameters on-line between simulations. The values of interface variables and internal variables may, of course, vary with time. The time derivative $\dot{x}$ of a variable x is written as "x'der." The scope rules for variables are very simple because of the powerful connection concept. Variables can be referenced only from equations in the submodel where they are declared.

It should be noted that the basic concept is not assignment statements but general equations. Thus, a model can be represented as $f(t, \dot{x}, x, p) = 0$. Many integration algorithms and simulation packages require that the derivatives are solved explicitly by the user: $\dot{x} = F(t, x, p)$. The support of general equations simplifies the model development, and the documentation becomes better since equations are closer to first principles. When developing a model for a physical system, one uses fundamental laws such as mass balances, energy balances, and phenomenological equations. These are either algebraic equations or ordinary differential equations that relate certain variables. Compared with the assignment form, it is easier to check that the model is entered correctly. The risk of introducing errors during manual transformation into assignment statements is reduced.

Furthermore, as thoroughly motivated by Elmqvist [2], the equation form is the only reasonable representation for model libraries. With models in assignment form, for each submodel, it must be decided which of its variables are inputs (in other words, are known) and which of its variables are outputs (defined by the model). As a simple example, consider a resistor. Ohm's law states $V_1 - V_2 = RI$, where V_1 and V_2 are the voltages at the ends of the resistor, I the current through the resistor, and R the resistance. The model has three variables: V_1, V_2, and I. In this model, the resistance R is a given parameter. If we should write the model in assignment form, there are three possibilities:

$$I := (V_1 - V_2)/R$$

$$V_1 := V_2 + RI$$

$$V_2 := V_1 - RI$$

The first variant assumes that V_1 and V_2 are inputs and defines I. This model is appropriate if, for example, one end of the resistor is connected to a voltage source and the other end is connected to the ground. The second and third variants assume that the current and the voltage at one end are known. These models are appropriate if the resistor is connected to a current source and the ground. Consequently, for models in assignment form, several different models are required for a resistor, depending on how it is connected to the environment. This makes both use and maintenance of a model library messy. Furthermore, other environments may result in algebraic loops so that equation systems with equations from several submodels must be solved to transform the model into assignment form. Two resistors connected in series between a voltage source and ground is a simple example of this. Submodels cannot be transformed into assignment form individually, since the transformation is a global problem.

A more sophisticated connection mechanism can be introduced when equations are allowed. Hibliz supports two types of connection semantics, depending on the character of the interaction variable. Consider, for example, three connected electrical wires. Each wire is represented by a voltage and a current. The constraints at the junctions are that the voltages are equal and the sum of the currents is zero. These two types of cut variables are sometimes called *across variables* or *through variables*. Other examples of across variables are pressure and temperature. Mass and energy flow, thrust, and torque are examples of through variables. It is natural to associate a direction with a through variable. The directions are important when putting up the equation for connected variables so that the terms in the "zero-sum equation" are given correct signs. The interfaces are presently declared textually; for example, the interfaces of a resistor may be declared as

Wire1: (V1: real, I1: IN real)

Wire2: (V2: real, I2: OUT real)

An interface variable is defined to be a through variable by the keyword IN or OUT as indicated above for I1 and I2.

Hibliz interprets the connections drawn by the user and generates appropriate equations automatically.

How Hibliz Is Operated

The user operates Hibliz via a keyboard and a mouse with three buttons. He can create and edit hierarchical block diagrams, inspect the model, and simulate it.

Scrolling, Panning, and Zooming

The mouse is used for pointing at the screen; a cursor follows the movements of the mouse. As indicated earlier, the mouse is used when scrolling, panning, and zooming. In order to scroll and pan, press the right button and move the mouse. To zoom, first press the right button and then the middle button. Now, moving the cursor up means zooming in; down means zooming out. An object (model, interface, curve, or text) can be zoomed in by pointing at it and pressing the middle button only. Zooming is done smoothly, and the final size of the object is chosen as large as possible while still being contained in the window. Text objects are treated specially. Their final sizes are chosen so that the longest line of the text matches the width of the zoom window. The text line pointed at is scrolled to the center of the zoom window. This allows convenient scrolling within a window. A connection is considered to belong to the enclosing model. Therefore, pointing at a connection is a convenient way of zooming out to a higher hierarchical level. If the middle button is pressed without pointing at anything, a zoom-out to 70 percent is performed. Note that it is possible to point at an object in a window other than the current zoom window. Pointing in an overview window thus allows rapid inspection of different objects. A new window is selected to be the current zoom window by pointing at it and pressing the left and right buttons.

Two-Button Stretching

To lay out objects such as windows, models, and interfaces on the screen is a common operation. The layout with respect to position and size is done with "two-button stretching." The lower left corner and the upper right corner of the objects follow the mouse in different ways depending on which of the left and middle buttons are pressed. If the left button is pressed, the cursor points at the lower left corner, which follows the movement of the mouse. The same applies for the upper right corner when the middle button is pressed. If both buttons are pressed, the cursor points at the center of the object, and the whole object moves. The stretching is finished when both buttons are released. It should be noted that the objects are completely redrawn, even in the two-button-stretching mode.

Commands

Commands are chosen from a pop-up menu. The menu is shown when the left button is pressed. The desired command is selected by pointing at the corresponding menu entry. Hibliz highlights the selected entry and performs the command when the button is released. If no entry is selected when releasing the button, the menu disappears. Command actions sometimes require additional input from the keyboard or mouse. Hibliz then prompts in the command area at the bottom of the screen.

The *Model command* creates a model and its graphical representation (two rectangles and the name). Hibliz prompts for name, which the user should type on the keyboard, and the enclosing model, which the user should select by pointing at it with the mouse and pressing the left button. The layout is done with two-button stretching as described earlier.

The *Interface command* is used to declare and position an interface. An interface is presently declared textually, and its graphical layout is done automatically. The user positions and stretches it using two-button stretching.

The *Connect command* makes it possible to draw connections between interfaces by using the mouse and the left button to input a sequence of line segments. The start interface is first selected by pressing the left button. Intermediate points are given by releasing and pressing the button. The last line segment is refreshed while the mouse is moved (rubber-band drawing), and the interface structure is searched for the destination interface.

The *Text command* is used to edit the textual parts of models. The editing is performed using a simple screen-oriented editor. When the user leaves the editor, the text description is parsed. Error messages are currently given as text in the command window. Here there are many possibilities to use graphics and color to explain the error to the user.

The *Remove command* allows deletion of models, interfaces, and connections. The *Copy command* copies a model and all its submodels, interfaces, connections, and the text. The *Save command* stores the current model hierarchy as a text file. Such a file can be read by the *Get command* to recreate the model. If a model has been defined earlier, the hierarchical position and layout are given as for the Model command. The Copy, Save, and Get commands make it possible to build simple model libraries.

The *Layout command* is used for changing the position and size of windows and models. There is no facility to change the layout of interfaces and connections. The *View command* creates a new window for viewing the model.

The *Compile command* analyzes the model and prepares for simulation. The *Simulate command* prompts for start and stop time and then starts the simulation.

The *Display command* creates a display for presentation of simulation results of any variable. The presentation of simulation results is very primitive at present. Only simple trend curves are implemented.

The *Hardcopy command* creates a description of the current content of the screen in the form of a PostScript program. The program can then be used to create hard copies on, for example, Apple's LaserWriter.

The *Exit command* stops Hibliz and returns to the operating system.

Implementation

The code for Hibliz consists of some 28,000 lines of Pascal. The software is highly modular. Related types, variables, and procedures are grouped together. Machine-dependent parts such as file and string handling are isolated to improve portability. A preprocessor, which we call Packman, is used to produce a standard Pascal program from module files, since Pascal does not allow mixed declarations of global constants, types, variables, and procedures.

Packman

Packman accepts files consisting of sections of code preceded by headings such as .PROGRAM, .LABEL, .CONST, .TYPE, .VAR, .FORWARD, .PROCEDURE, .INIT, and .MAIN. Packman outputs the contents of all sections labeled .CONST on a file named CONST.SEC, and so on.

Typically, the file issued to Packman is a short file containing a number of commands to include files. Inclusions may be nested. Hibliz is built similar to a transparent onion consisting of six layers. Outer layers can use elements of inner layers. Packman is designed to promote separate compilation. By the commands .DEFINITION or .IMPLEMENTATION it is possible to specify what should be visible outside the compilation unit and what should be hidden. Portability with respect to separate compilation facilities of different Pascal compilers can be handled by modifying Packman.

Data Structure

A model is represented as a record containing lists of submodels, variable declarations, connections, and equations. A general list package for doubly linked lists with headers is used. It has operations such as NewList, Into, First, and SuccElem. A *node* is a Pascal record with variants. The common area contains information such as forward and backward pointers for list manipulation; the variant part contains a pointer to a record describing models, interfaces, connections, etc. Note that this method of implementing lists makes it possible for a model description, etc., to be a member of several lists at the same time. The list package makes it easy to handle and manipulate lists.

Compiler

The model descriptions at the lowest level are parsed at exit from the editor or when read from a file by the Get command. The parser builds a syntax tree for each equation. The Compile command links the identifiers to their declarations. Interface variables connected to each other are put into a list. These lists of connected interface variables are then analyzed to generate the proper equations, their syntax trees, and the links to declarations. It also checks that across and through variables are not connected to each other. The type consistency of all expressions is checked.

When Hibliz has collected all equations, it has a differential/algebraic system of the form

$$g(t, \dot{x}, x, v, p, c) = 0$$

where t is the time, x and v vectors of unknown variables, p a vector of known parameters, and c a vector of known constants. The vector v contains those unknown variables that do not appear differentiated in the equations. Hibliz uses the differential/algebraic system solver (DASSL) [3]. DASSL has a reputation of being one of the best and most robust numerical solvers for differential/algebraic systems. DASSL accepts problems of the above form if it is provided with a routine for calculating the residual $\Delta = g(t, \dot{x}, x, v, p, c)$ when the arguments are known. However, to decrease the order and complexity of the problem, simple symbolic formula manipulation is performed as follows. Connections of across variables lead to simple identities of the form $A = B$. It is easy to explore these entities and eliminate variables. The record describing a variable has an element called *alias*, with an initial value implying that it is its own alias. When simple equations are found, Hibliz modifies the alias elements accordingly and removes the equation from the list of interesting equations.

After the elimination of simple equations, Hibliz assumes that $\dot{x}$ and v are unknown and sorts the equations and variables so that the problem becomes block lower triangular with minimal diagonal blocks. If a block is scalar and the variable to be solved from the equation is a component v_i of the vector v, and the equation is of the form $v_i = \langle$expression independent of $v_i\rangle$, the variable v_i is eliminated from the vector of unknown variables passed to DASSL. The routine for evaluating the residual Δ can, in this case, calculate v_i itself. The partition to lower block triangular form may fail. An error message is then given listing unassigned variables and redundant equations. The problem is then either singular or has algebraic relations between the components of the vector x.

To make the calculation of the residual vector Δ more efficient, Hibliz generates code for a virtual stack machine. The code is interpreted by a Pascal procedure. The values of constants, parameters, and variables are stored in a global array.

The *Run command* sets the initial values of states as given by the initial section of the models and then uses DASSL to solve the system.

Graphics

Routines for handling graphics are an essential part of Hibliz. A local coordinate system is assigned to each model such that the lower left corner of the rectangle has the coordinates (0,0) and the upper right corner has the coordinates (1,1). The positions of its interfaces, submodels, equations, etc., are expressed in this coordinate system. When moving and scaling a model, this hierarchy of coordinate systems makes it almost trivial to scale and move its submodels properly. All coordinates are stored as real numbers since continuous zooming requires high-resolution coordinates.

Hibliz currently runs on an IRIS 2400 from Silicon Graphics, Inc. [4]. The IRIS is a high-performance engineering workstation designed for interactive color graphics and computing applications. The program interface to the graphics is the IRIS Graphics Library. It has routines for definition and manipulation of objects in (local) world coordinate systems and projection of these objects onto the screen. The IRIS has special graphics hardware for transformations from local world coordinates into screen coordinates. Clipping and scan conversion also are done in hardware.

Fonts

A block diagram contains text, and to make continuous zooming possible it is necessary to display characters of different sizes. The IRIS Graphics Library supports one fixed-pitch raster font of height 16 and width 9 pixels. New raster fonts can be defined; however, because of memory constraints, larger characters have to be viewed as graphical objects consisting of straight and curved lines.

The authors have developed a support program to generate new fonts. This program is based on ideas given in [5]. The user defines the shape of a character as a number of line segments and the size and form (rectangular, circular, oval, etc.) of the pen to be used. A line segment is defined by its start and end points and its tangents in these points. Intermediate points on the line segment are defined by a cubic spline function ([5], pp. 24–26).

Hibliz uses both raster and graphical fonts; for characters up to 16×20 pixels, raster fonts are used. The use of both raster and graphical fonts makes outputting of text somewhat more complex. There are, however, two good reasons for using raster fonts and not only graphical fonts. First, when drawing a small character, the quantization may deform the character so that it looks distorted and is difficult to recognize. If a character is moved over the screen, its form changes due to the quantization, and, for example, an "o" looks like an amoeba. Second, it is important to make the graphics as fast as possible. When the characters are small there may be many of them on the screen. If the characters are defined as graphical objects, this implies drawing of many line segments. For example, to draw a nice-looking O, at least 20 line segments are needed. The IRIS can draw a maximum of 65,000 line segments per second; whereas, it can display up to 150,000 raster characters per second.

Conclusions

Some of the ideas on graphics presentation and interaction have been further carried through by the first author at SattControl. That has resulted in a product called Sattgraph 1000 [6], which is a presentation system for plant operators. Sattgraph 1000 uses the concept of information zooming to deal with hierarchical structuring and has an object-oriented approach to interaction during creation of pictures and for operating a plant.

The structural properties of a model are very important, particularly when working with large, complex systems. It is the authors' belief that it is easier to describe structural properties when graphics are used than when a purely textual description must be used. In this paper, the authors propose the use of hierarchical block diagrams, which can be scrolled, panned, and zoomed continuously. The block diagram describes the model decomposition and the interconnection structure. The zooming controls the amount of information displayed. When zooming in on a block, the block changes from an annotated box into a representation showing the internal structure with increasing detail. Since the block diagram can be made hierarchical, it is possible to make the description at each level simple and clear.

The new workstations with fast, high-performance graphics make it possible to implement the man-machine interface proposed. To demonstrate the feasibility of the proposal, a prototype system, called Hibliz, has been implemented. Hibliz also simplifies model development by allowing submodels in the form of ordinary differential and algebraic equations rather than assignment statements for derivatives and algebraic variables.

Acknowledgments

The authors would like to thank Professor Karl Johan Åström for many useful discussions and for his strong support. Many thanks to Tommy Essebo and Dag Brück, who have programmed parts of the system. This work has been supported by the National Swedish Board for Technical Development under Contract 84-5069.

References

[1] H. Elmqvist, "LICS—Language for Implementation of Control Systems," Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, Rept. CODEN: LUTFD2/TFRT-3179, 1985.

[2] H. Elmqvist, "A Structured Model Language for Large Continuous Systems," Ph.D. thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, Rept. CODEN: LUTFD2/TFRT-1015, 1978.

[3] L. R. Petzold, "A Description of DASSL: A Differential/Algebraic System Solver," *Proc. IMACS World Congress*, Montreal, Canada, 1982.

[4] J. H. Clark and T. Davis, "Workstation Unites Real-time Graphics with Unix, Ethernet," *Electronics*, pp. 113–119, Oct. 20, 1983.

[5] D. E. Knuth, *T_EX and Metafont—New Directions in Typesetting*, American Mathematical Society and Digital Press, 1979.

[6] H. Elmqvist, S. O. Eriksson, and S. Runeson, "New Concepts for Operator Interaction Using Modern Graphic Techniques," Programmable Controllers '86 Conf., Birmingham, Nov. 1986.

Hilding Elmqvist received his master's degree in 1972 at the Lund Institute of Technology, Lund, Sweden. He received his Ph.D. degree from the Department of Automatic Control, Lund Institute of Technology, in 1978. The thesis treated a language for modeling of large systems. He then decided to get more involved in computer science and spent a postdoctoral year in 1978–1979 at the Computer Science Department at Stanford University, California. He then returned to Sweden, where he returned to automatic control and started a project on Languages for Implementation of Control Systems. In 1984, he joined the company SattControl, which manufactures control systems. He got the opportunity to realize his ideas further there, and that has so far resulted in the product SattGraph 1000, which is a presentation system. He is now Manager of the Software Systems Department, which works on a totally integrated distributed control system.

Sven Erik Mattsson received the Ph.D. degree from the Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, in 1985. His doctoral work was on modeling and control of large horizontal axis wind power plants. Dr. Mattsson currently is conducting a research project to develop prototype tools for model development and simulation. His research interests include development and application of tools for computer-aided control engineering.

An Intelligent Help System for Idpac

Jan Eric Larsson M.Sc. Lic.Tech. & Per Persson M.Sc. Lic.Tech.

Department of Automatic Control
Lund Institute of Technology
Box 118, S-221 00 LUND, Sweden

Abstract

This paper describes an expert system interface for a program for system identification. It works as an intelligent help system for the interactive program Idpac, using the *command spy* strategy. This means that the system is completely non-invasive and uses the previous command history to understand what the user is doing and gives help according to this. *Scripts* are used for representing procedural knowledge and production rules for diagnostic knowledge. The system has been implemented and a knowledge database developed. An example run with the system is shown.

Introduction

A modern CAD program usually is quite complex and demands the user to have a lot of knowledge, about the program as well as about the problem domain. For this reason, there is a need for help systems with knowledge about both these areas. We believe that an expert system is well suited for the implementation of such a help system. In order to use expert system techniques in a CAD program, several problems must be solved.

- CAD programs usually have a flexible command dialog. This way of communication should be retained when the expert system is added to the program.
- The expert system should be totally non-invasive, allowing the user to fall back on the plain CAD program in case it is not able to give the user any help.
- An inexperienced user often has a general idea of what he wants to do, but does not know exactly how to do it. The expert system should be able to guide the user from general ideas to specific commands, i.e., it should give goal related help.
- An intelligent help system should have facilities to teach the user about the target program and to transfer knowledge from the knowledge database to the user.
- An expert system used as an intelligent help system interface must be able to handle both procedural and diagnostic knowledge. We propose to represent procedural knowledge with scripts, and diagnostic knowledge with production rules.

According to these design criteria an expert system interface has been developed. It contains a command parser, a script matching device with a database for scripts and rules, a query module, a file system, an on-line dictionary, and interfaces to the user and Idpac.

We have focussed the project on the idea of a non-invasive, goal related help system in general, and not in any special kind of man-machine communication. Thus, the decision was made to use almost the same command language in the interface as in Idpac. An alternative would have been to equip the system with, e.g., a graphics interface, but as this was not necessary for the development, we abstained from the effort.

Idpac is a command driven program for system identification, see [1,2,3]. This project was originally outlined in [4,5]. A previous system was described in [6]. A thorough description of this project is given in [7,8,9]. There are some other projects in the area of intelligent help systems, see [10,11].

System Identification

System identification is the process of finding a mathematical model of a real world process. The models can be differential or difference equations, relating input signals to output signals. System identification involves experiments in which input and output signals of a physical process are measured, choice of a suitable model, validation of data, numerical fitting of model parameters to these data, and several validation tests for verification of the results. System identification demands skill and experience and the validity of the results strongly depends on the user's knowledge. For thorough readings on system identification, see [12,13]. An overview is found in [14].

Scripts

Running Idpac requires a lot of procedural knowledge, i.e., knowledge about sequences of commands, and, on a higher level, knowledge of sequences of subtasks of Idpac sessions. In order to represent sequences, the concept of *scripts* was introduced. We were inspired by a data structure used in natural language understanding, [15,16]. It should be noticed that our script concept is different from Shank's. We use scripts to describe the possible order of a number of commands, where the commands must have certain attributes, e.g., parameters, in order to match. Scripts may be implemented in several ways, e.g., with production rules or as Lisp lists. The latter approach was taken in this project. Here is an example of a script.

```
((command plot (infile INSI) (infile OUTSI))
 (repeat
  ((command mlid (outfile SYST) (infile INSI)
                 (infile OUTSI) (number N))
   (kscall (Estimation of order (parameter N) performed))
   (or
    ((command residu (outfile RES) (infile SYST)
                     (infile INSI) (infile OUTSI)))
    ((command sptrf (outfile FREQ) (infile SYST))
     (kscall (Recommend multi-Bode plot with (parameter FREQ)))
     (command bode (infile FREQ))))))
 (kscall (Give advice on most probable order)))
```

This script describes a way to perform a parameter estimation with increasing order of the fitted model. First the user should look at the signals with the PLOT command. The input files corresponding to INSI and OUTSI should already exist. A model is produced with MLID. The output file SYST is created and N, the order of the model, is associated with the actual argument used. Next, a fact stating that a parameter estimation has taken place is sent to the production rule database. This is done with the KSCALL clause. Kscall stands for Knowledge Source Call and puts a fact in the fact database of the production rule system, which is associated with each script. After this, the user may either look at the residuals with the RESIDU command, or produce data for a Bode plot with SPTRF and plot it with BODE. This is expressed with the OR clause. The REPEAT clause means that this whole procedure may be repeated, and every time the rule system gives advice on whether the order is sufficiently high. During the process it may use facts put into the database by previous kscalls. This script is of course far too small to be realistic, but it shows what a script may look like. For a script of reasonable size, see [9].

Other clauses in the script language are the ALL clause, which expresses that all the following commands must match, but the order of them is not essential, and the SCRIPTMACRO clause, which is a kind of subroutine.

The Knowledge Based Command Spy

How does one combine a CAD program with an expert system while keeping the good features of both? The solution proposed in this paper is the *command spy* strategy.

The expert system is used as an interface to Idpac. In our solution it is placed before the command decoder of Idpac, but an alternative would be to build it into the outermost level of Idpac.

We keep the command language of Idpac. In this way a cumbersome Q/A dialog is avoided. The expert system traces the user without asking any questions, and gives help only on demand. Thus the expert system never forces a user to do anything. A user that does not need or want any help is not bothered and there is always the possibility of falling back on plain Idpac in case the help system does not have enough knowledge to work properly. The expert system may also use a Q/A dialog to find out facts about the experiment and expected results, but only if the user initiates it.

The command spy uses scripts in order to understand what the user is doing. By matching scripts against the actual command history, the expert interface is able to guess what the user wants to do. The scripts also provide information on the next step for reaching a desired goal.

Diagnostic reasoning is needed at certain points in an identification, notably when things go wrong or unexpected results arise. This is taken care of by production rules associated with each script. The rules also allow for automatic documentation by writing script based information to a text file.

Several scripts may be active at the same time, as long as they match the commands typed by the user. This is typically the case when the system is started. At any time the user may ask for the next sensible command. The command spy then looks at the next possible commands in all the active scripts, and gives these commands as the answer. If, however, the user does not follow this advice, some error recovery actions are taken. One possibility is to assume that the user wants to start all over again, so the initial scripts are tried once more. If one of them should match, the current script gets suspended and the new script becomes the current script. If the user's command does not match any of the initial scripts, the command spy checks whether the command matches any previously suspended scripts. If so, the current script gets suspended and the script which matched the command becomes the current script. If none of this is the case, the command spy stays in its current state, sends the, for the command spy unintelligible, command to Idpac, and from the next command on it tries to restart again.

Implementation

The expert interface is made up from several independent parts. Most of the parts work on a common database.

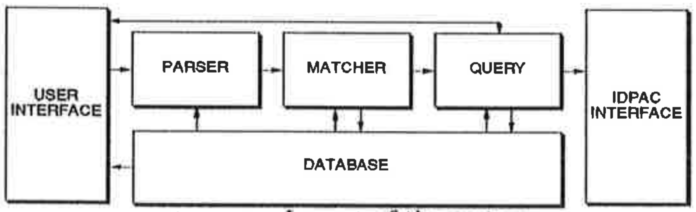


Figure 1. Layout of the system.

The user interface module reads a command from the user and transforms it into a Lisp list. It provides all input and output functions used in other parts of the interface. In this way, all of the system's dependence on terminal types, graphics, etc., is collected in one place.

The command parser module checks the command for syntax and supplies defaults in the same way that the parser of Idpac does. In this process it transforms the command into a more convenient form. This parser accepts commands with the arguments left out, as the other routines will fill information in, by defaulting from scripts and asking the user.

The script matcher module keeps track of the scripts incrementally and updates them as it gets commands from the parser. The commands once again are transformed, and files may be defaulted using knowledge from the scripts.

The query module works through the command description and tries to fill in the remaining unknown entries by asking the user about them. In this way the user may give only the command name. If necessary, he will then be prompted for any parameters left out. The query module also sends messages to the file system about created files.

The database contains the command grammar used by the parser, the scripts and rules used by the script matcher, the file tree of the file system, and state variables for keeping track of the user state, set a debug mode, etc.

The file system keeps track of all the data files created and used during an Idpac session. It does this by storing data about the files in a directed graph structure. This enables the file system to show, e.g., the ‘ancestors’ or ‘descendants’ of a file.

The expert interface and Idpac reside in two different VMS processes. The Idpac interface sends the processed command to Idpac via a VMS mailbox. In this way no changes had to be done to the Idpac program itself. The interprocess communication routines are written in C.

The system is written in Franz Lisp extended with Flavors [17], and YAPS [18]. It consists of about 6000 lines of code and runs under VMS on a VAX 11/780.

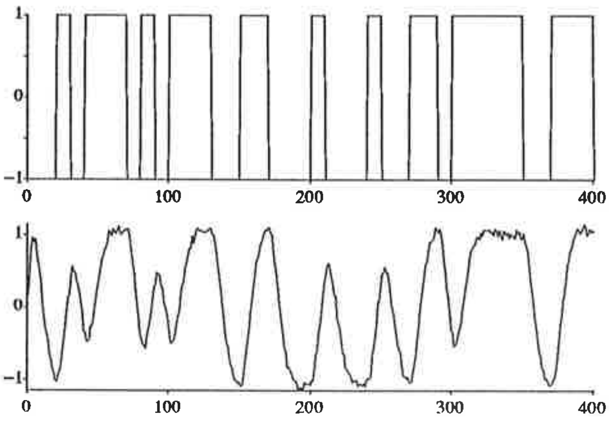
The Knowledge Database

A knowledge database has been constructed during the project. It only deals with parameter estimation using the maximum-likelihood method, and thus, it does not cover everything that Idpac may be used for. Still, we believe that this small database shows that our solution will indeed work. The system is used for demonstrations and has also been used in an undergraduate level course on system identification. The database is thoroughly described in [9]. The script is about 300 lines long, and the rule base contains about 200 rules. This script and these rules can reasonably well handle the the interactive session of a student.

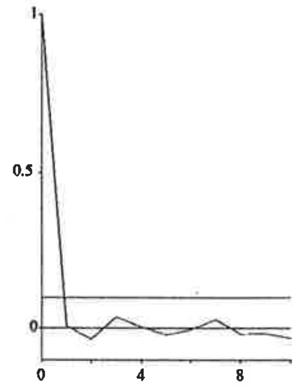
An Example Run

This is an example of a run with the system. The script used is the one shown in the script section of this paper. Approximately 10 rules are needed to handle this script. A ‘?’ asks for rule based information and ‘??’ for the next command according to the script. Texts appearing after a ‘!’ are comments.

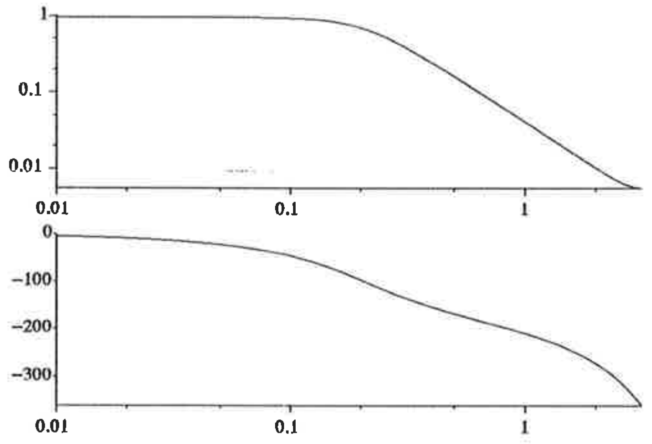
```
>plot in / out
```



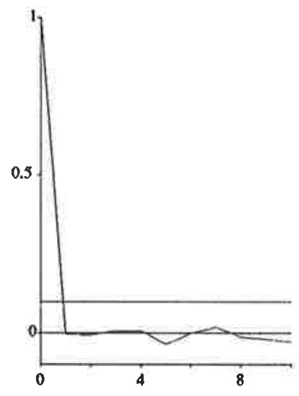
```
>??
Next command: mlid
>mlid syst2 < in out 2        ! Create a second order model.

 A-POLYNOMIAL
 1. - 1.71073Q^-1 + 0.747169Q^-2

 B-POLYNOMIAL
 Q^-1 * (8.10165E-3 + 2.73358E-2Q^-1 )

 C-POLYNOMIAL
 1. - 0.925355Q^-1 + 0.303477Q^-2

>?
Look at the residuals of in and out, using the system syst1.
>residu                       ! The user gives no arguments.
 residu ... < syst2 in out   ! Some are defaulted.
 Residuals outfile? > res2   ! The system prompts for the rest.
```



```
>??
Next command: mlid or sptrf
>sptrf freq2            ! Compute a transfer function.
>bode                   ! Plot it.
```

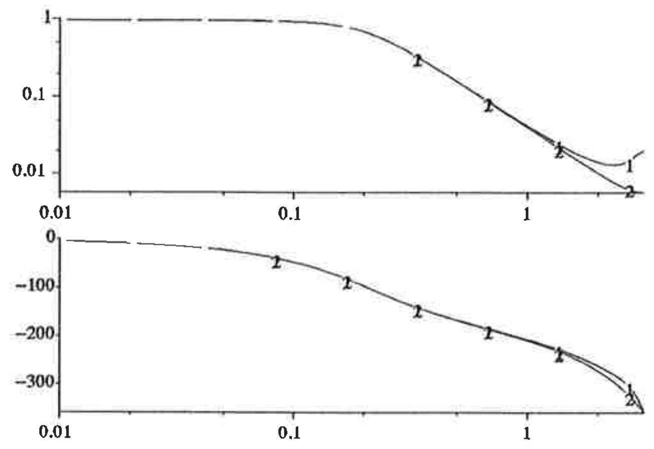


```
>ml syst3               ! Create a new model.
 mlid syst3 < in out ...! No default for model order.
 Model order? > ?       ! The user wants a further explanation.
 The order of the transfer function to be fitted to the indata.
 Model order? > 3

 A-POLYNOMIAL
 1. - 1.02007Q^-1 - 0.43999Q^-2 + 0.521125Q^-3

 B-POLYNOMIAL
 Q^-1 * (5.14744E-3 + 4.01447E-2Q^-1 + 1.38089E-2Q^-2 )

 C-POLYNOMIAL
 1. - 0.221032Q^-1 - 0.375885Q^-2 + 0.245745Q^-3

>res res3
```



```
>??
Next command: mlid or sptrf
>sptrf freq3 < syst3 b / a     ! Compute a transfer function.
>?
Perform the command BODE freq3 freq2.
```

```
>bode freq3 freq2
```



```
>?
The model order is probably high enough when the Bode plots
start to coincide in the medium frequencies.
```

Conclusions

The main conclusion of the project is that if one wants to combine an expert system with a CAD program, the expert system should be used as an intelligent and non-invasive help system. This retains the advantages of both the CAD program and the expert system. It may be accomplished by implementing a command spy, as outlined in this paper. Secondly, not all knowledge need be implemented with production rules. Scripts may be a better way of representing sequences, especially in problems where both methods and goals are well known. A good rule is to use as much as possible of the structure of the problem in its solution. The use of scripts will also reduce the size of the knowledge databases considerably.

Acknowledgements

We would like to thank our supervisor Karl Johan Åström and our colleagues Sven Erik Mattsson, Lars Nielsen, and Karl-Erik Årzén for supporting us in our work.

The project has been part of the Computer-Aided Control Engineering project at the Department of Automatic Control, Lund Institute of Technology, supported by STU, the National Swedish Board for Technical Development, under contract no. 85-3042.

References

1. Wieslander, J., *Interaction in Computer-Aided Analysis and Design of Control Systems*, Doctorial dissertation, TFRT-1019, Department of Automatic Control, Lund Institute of Technology, 1979.
2. Gustavsson, I. and A. B. Nilsson, "Övningar för Idpac," Internal report, TFRT-7169, Department of Automatic Control, Lund Institute of Technology, Lund, 1979.
3. Gustavsson, I., "Några macros för Idpac,", Internal report, TFRT-7170, Department of Automatic Control, Lund Institute of Technology, Lund, 1979.
4. Larsson, J. E., *An Expert System Interface for Idpac*, Master thesis, TFRT-5310, Department of Automatic Control, Lund Institute of Technology, 1984.
5. Larsson, J. E. and K. J. Åström, "An Expert System Interface for Idpac," *Proceedings of the 2nd IEEE Control Systems Society Symposium on Computer-Aided Control System Design*, Santa Barbara, California, 1985.
6. Larsson, J. E. and P. Persson, "Knowledge Representation by Scripts in an Expert Interface," *Proceedings of the 1986 American Control Conference*, Seattle, Washington, 1986.
7. Larsson, J. E. and P. Persson, *An Expert System Interface for Idpac*, Licentiate thesis, TFRT-3184, Department of Automatic Control, Lund Institute of Technology, Lund, 1987.
8. Larsson, J. E. and P. Persson, "The (ihs) Reference Manual," Technical report, TFRT-7341, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1987.
9. Larsson, J. E. and P. Persson, "A Knowledge Database for System Identification," Technical report, TFRT-7342, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1987.

10. Wilensky, R. *et al*, "UC—A Progress Report," Report No. UCB/CSD 87/303, Computer Science Division (EECS), University of California, Berkely, California, 1986.

11. Waters, R. C., "KBEmacs—A Step Toward the Programmer's Apprentice," Technical report 753, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1985.

12. Eykhoff, P., *System Identification. Parameter and State Estimation.*, John Wiley & Sons, London, 1974.

13. Eykhoff, P., *Trends and Progress in System Identification*, Pergamon Press, Oxford, 1981.

14. Åström, K. J., "Modeling and Simulation Techniques," Agard Lecture Series No. 128, 1983.

15. Schank, R. C. and R. P. Abelson, *Scripts, Plans, Goals and Understanding*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1977.

16. Schank, R. C. and C. K. Riesbeck, *Inside Computer Understanding*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1981.

17. Allen, E. M. *et al*, "The Maryland Artificial Intelligence Group Franz Lisp Environment," TR–1226, Department of Computer Science, University of Maryland, 1984.

18. Allen, E. M., "YAPS—Yet Another Production System," TR–1146, Department of Computer Science, University of Maryland, 1983.

CODEN: LUTFD2/(TFRT-3186)/1-018/(1987)

Combination of Symbolic Manipulation and Numerics

Ulf Holmberg
Mats Lilja
Sven Erik Mattsson

STU project 85–4809

Department of Automatic Control
Lund Institute of Technology
March 1987

Department of Automatic Control **Lund Institute of Technology** P.O. Box 118 S-221 00 Lund Sweden	*Document name* Report
	Date of issue March 1987
	Document Number CODEN: LUTFD2/(TFRT-3186)/1–018/(1987)
Author(s) Ulf Holmberg Mats Lilja Sven Erik Mattsson	*Supervisor*
	Sponsoring organisation The National Swedish Board of Technical Development (STU contract 85-4809)

Title and subtitle
Combination of Symbolic Manipulation and Numerics

Abstract

Symbolic manipulation will play an important role in future tools for Computer Aided Control Engineering (CACE). The purpose of this project has been to experiment with and gain experiences of using a program for symbolic manipulation. As the tool we used the symbolic manipulation program MACSYMA and as the application analysis of multivariable systems was selected. A small package for analysis of multivariable linear systems is constructed inside MACSYMA. Manipulations of polynonial matrices are given as example. Also it is shown how it is possible to transfer results from MACSYMA to numerical programs, like the matrix-manipulation program CTRL-C and the differential equation program Simnon. By combining symbolic and numeric computations we can solve more complex and composit problems. This is demonstrated by solving for multivariable root locus.

Key words
Symbolic Manipulations; MACSYMA; Computer Aided Control Engineering; Multivariable Linear Systems; Polynomial Matrices.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title		*ISBN*
Language English	*Number of pages* 18	*Recipient's notes*
Security classification		

1. Introduction

Symbolic manipulation will play an important role in future CACE tools. Unfortunately, today's systems for Computer Aided Control Engineering (CACE) allow basically the user to perform numerical calculations. They do not support symbolic calculations. One important reason for this is that the tools were designed for computers with what is today considered as moderate computing power and symbolic manipulation calls for computing power. The increasing capacity of computers and workstations makes it now worthwhile to introduce symbolic calculations in CACE systems.

[Pavelle et al., 1981] give a popular scientific introduction to computer algebra. There are commercial general-purpose systems for symbolic systems available:

MACSYMA	developed at the MIT Laboratory for Computer Science, USA
REDUCE	developed at Stanford University, the University of Utah and the Rand Corporation, USA
Scratchpad	developed by IBM
SMP	developed at the California Institute of Technology, USA
Maple	developed at the University of Waterloo, Canada
muMATH	developed by the Soft Warehouse, Honolulu

The main purpose of the project "Combination of Symbolic Manipulation and Numerics" was to experiment with and gain experiences of using a program for symbolic manipulation. As the tool we used MACSYMA and as the application analysis of multivariable systems was selected. There is a framework for analysis and design of multivariable systems using polynomial matrices. A standard text book is [Kailath, 1980]. Unfortunately, these methods have poor numerical properties. Methods based on state space representations have better numerical properties. However, in many cases it is desirable to be able to work in the frequency domain. It may be easier to formulate and analyse properties of interest in the frequency domain than in the state space. MACSYMA is good at polynomials and rational functions.

Motives for supporting symbolic calculations are presented in Chapter 2. The package developed in MACSYMA for analysis of multivariable linear systems is described in Chapter 3. A Lisp function in MACSYMA can be used to establish interaction with other programs, such as Simnon and CTRL-C. This is presented in Chapter 4. In Chapter 5 a new method for calculating root loci is demonstrated. This serves as an example of the idea to combine symbolic and numerical computation and thereby solve more complex and composite problems. Conclusions are given in Chapter 6.

2. Motives

There are several good reasons for including symbolic manipulation in CACE tools. First, structure is important and an analytic answer may give a better insight. Second, the user interface could be improved, since the user's original formulation is usually not a computational procedure but rather equations and relations on symbolic form. Third, models could be multi-purpose and reusable independent of what is known and what should be computed. Forth, symbolic manipulation could be used to facilitate numerical solution. Below we will discuss these motives in more detail.

Insight is Desirable

You may think that the ultimate CACE program is an automatic procedure which outputs a VLSI chip that implements the optimal controller. Life is not that easy. You must at least specify your desires and requirements; a specification of what **you** think is optimal. Unfortunately, this may be a laborious and demanding task. Many problems are not that well-defined. If it is a new type of plant, it might be difficult to know which are the decisive requirements and which that are easy to fulfil. A given constraint may be totally decisive for the outcome of the control design. A designer may be willing to adjust the requirements to achieve other benefits, but he is not willing to consider every case or combination. He wants to work in an iterative way and be able to eliminate bad approaches early. He also wants to know why a certain approach fails. He wants to get insight. For example, it may be easy and favourable to remove a constraint by redesigning the plant. It is in most cases favourable to take the interaction between process design and control design into account and consider them simultaneously.

A designer is happy when he has a profound understanding of the system dynamics. He then knows the possibilities and the limitations and can make the proper compromises during the design. He can justify why it is not possible to make a better design according to the circumstances. In many cases insight is the key to design. If you can pinpoint the critical parts and if you understand the difficulties, you can often solve or avoid the problems and make a good design.

In real life most plants have significant non-linear behavior, while most available software for analysis and synthesis assumes linear models. It is difficult to analyse non-linear systems. The simulation model could be used for empirical studies concurrently with a mathematical analysis. Possibilities to include and exclude different features in the model by changing the model for one part or by making parameter changes are useful when studying their importance. To have some success with the analysis we are more or less forced to work mainly with linear models and to estimate the effects of nonlinearities. Linearization is tedious to do with paper and pen. A good formula manipulation program which takes the nonlinear equations and outputs the linearized ones would be a real time-saver. If there also was a program that took the linear model and intervals for the parameters and made proper approximations, the analysis would be even simpler to carry out. A nice thing with linear models is that they can be transformed into the frequency domain where many dynamical properties are easier to understand. When analysing a system it is useful to have different viewpoints and possibilities to transform back and forth between different representations.

Support Users' Concepts

It is important that a user can describe his problems on a for him natural form. The user interface of a CACE system could be viewed as consisting of a language and environment. The language should be more than just a means for instructing the computer to perform tasks. It should also serve as a framework within which we organize our ideas. It should be a high level problem solving language.

It is important that the user can give the mathematical description of a submodel on a natural form. When deriving models from first principles the result is often a system of differential algebraic equations (DAE):

$$g(t, \dot{x}, x, v, p, c) = 0$$

where t is the time, x and v vectors of unknown variables, p a vector of known parameters and c a vector of known constants. It is natural to require that interactive software for model development and simulation supports DAE systems. The prototype simulator Hibliz ([Elmqvist and Mattsson, 1986]; [Mattsson, Elmqvist and Brück, 1986]) which was developed in another CACE project (STU project 84-5069) accepts mathematical descriptions given as DAE systems. However, most simulation packages of today do not allow models given as DAE systems, but require assignment statements for derivatives and algebraic variables. The user must solve for the derivatives and put the model on the form

$$\dot{x} = f(t, x, p, c)$$

He is often allowed to introduce sequences of auxiliary variables and to give the assignment statements in any order:

$$\begin{aligned} \dot{x} &= f_1(t, \dot{x}, x, v, p, c) \\ v &= f_2(t, \dot{x}, x, v, p, c) \end{aligned}$$

as long as it possible to sort them so that all derivatives and auxiliary variables are calculated before use. This means that the user has to manipulate his model manually. This is a non-trivial task. Errors may be introduced.

When DAE systems are supported, the model becomes more readable since the user can recognize fundamental relations as mass and energy balances and other phenomenological equations. It is easier to check that the model is entered correctly and the risk of introducing errors during manual transformation is reduced.

Multi-Purpose and Reusable Descriptions

It is a laborious and time-consuming task to develop good models of plants and various phenomena. Consequently, it is important that the investments in model development can be reused. A model can be used for different purposes as simulation, analysis and design. The status of a variable may vary. Sometimes it is considered to be known, while in other situations we want to solve for it. For example, when solving for a stationary operating point the derivatives are set to zero and the states are to be solved for. When a numerical ODE solver is used, the states are considered to be known and we should solve for the derivatives. When designing the plant or the control system, some of the parameters are considered to be unknown by the designer.

Furthermore, as thoroughly motivated by [Elmqvist, 1978], the equation form is the only reasonable representation for model libraries. With models on assignment form, it must be decided for each submodel which of its variables that are inputs (in other words are known) and which of its variables that are outputs (defined by the model). As a simple example consider a resistor. Ohm's law states $V_1 - V_2 = RI$, where V_1 and V_2 are the voltages at the ends of the resistor, I is the current through the resistor and R is the resistance. The model has three variables V_1, V_2 and I. The resistance R is in this model a given parameter. If we should write the model on assignment form there are three possibilities

$$\begin{aligned} I &:= (V_1 - V_2)/R \\ V_1 &:= V_2 + RI \\ V_2 &:= V_1 - RI \end{aligned}$$

The first variant assumes that V_1 and V_2 are inputs and defines I. This model is appropriate if for example one end of the resistor is connected to a voltage source and the other to ground. The second and third variants assume that the current and the voltage at one end is known. These models are appropriate if the resistor is connected to a current source and ground. Consequently, for models on assignment form we need several different models for a resistor, depending on how it is connected to the environment. This makes both use and maintenance of a model library messy. Furthermore, other environments may result in algebraic loops so that equation systems with equations from several submodels must be solved to transform the model into assignment form. Two resistors connected in series between a voltage source and ground is a simple example of this. Submodels cannot be transformed into assignment form individually, but the transformation is a global problem.

Improve the Numerical Properties

It is favourable if a CACE system accepts problems on forms preferred by users. By exploring symbolic manipulation the problems can in many cases be simplified and transformed to a form more suitable for numeric solution.

As an example consider the problem of finding the optimum of a function. The numerical solution procedure could be made faster and more robust if analytic procedures for calculating the gradient and the Hessian are given. However, in many cases it is laborious for the user to provide these procedures. It is much more convenient for him if they are generated automatically.

You may say that a problem is ill-conditioned if a small perturbation in the equations can lead to a large deviation in the solution. The main question is, however, what perturbations we have to consider in a particular case. If we have a fully parameterized model, where all explicit numbers are exact (structural ones, zeros etc), the perturbations of interest are those described by perturbations in the parameters. If we want to perform a numerical calculation and substitute the parameters with numbers, then it is of interest to consider unstructured and random perturbations to model for example quantization. Then a larger class of problems becomes ill-conditioned. For a fully parametrized problem the condition number is not a problem invariant, but it depends on the formulation and may be decreased by symbolic manipulation.

It is important to consider the structural properties of a problem when deciding whether it is well-posed or not. For example the problem $\varepsilon\dot{y}+y=1$ where we know that the system is stable, is well-posed. The only perturbations that we have to consider are perturbations in ε which lead to an ε greater or equal zero. Even from a numerical view, it must be considered to be well-posed. It is a minimum demand that a non-negative number is represented by the computer as a non-negative number.

Possibilities to use symbolic manipulation to handle DAE systems are discussed in [Mattsson, 1986].

3. An Analysis Package in MACSYMA

The framework of polynomial matrices is useful for analysis of multivariable linear systems, see [Kailath, 1980]. However, polynomial matrices are not easily manipulated by hand. It is thus very important that good analysis tools for polynomial matrices are available. We have tried to fill the gap between theory and practice by implementing a package for analysis of multivariable linear systems in MACSYMA. The functions of this package is listed below. Then a MACSYMA demo with matrix fraction decompositions, co-prime factorizations, multivariable realizations, etc., will illustrate the beauty of symbolic manipulations. The examples are taken from [Kailath, 1980]. For further examples and details on the implementation including listings of the functions we refer to [Holmberg, 1986].

Available Functions

The following functions for analysis of multivariable linear systems have been implemented in MACSYMA.

	Linearization
`LINEARIZE`	Linearizes the dynamical system $\dot{x} = f(x,u), \quad y = g(x,u)$
	Stability analysis
`ROUTH`	Generates the stability conditions for a continuous time system
`JURY`	Gives the stability conditions of discrete time systems and the steady state output variance
	Sampling
`SAMP`	Sampling from transfer function to pulse-transfer function
`SAMPSTATE`	Sampling from state space to state space
	Geometry functions — state space
`HERMITE`	Gaussian elimination when applied to a constant matrix
`KER`	Computes the Kernel $\{X \mid AX = 0\}$
`INVERSE_IMAGE`	Calculates the inverse image $\{X \mid AX = B\}$ (A possibly singular)
`INTERSECTION`	Computes the intersection of two subspaces
`GRAM_SCHMIDT`	Calculates an orthogonal base for a subspace
`AINV`	Computes the maximal A-invariant subspace in a given subspace
`ABINV`	Computes the maximal (A, B)-invariant subspace in a given subspace
	Factorization — Frequency domain
`SMITH`	Calculates the Smith form together with transformation matrices
`SMITH_McMILLAN`	Calculates the Smith-McMillan form with transformation
`HERMITE`	Calculates the Hermite form
`COLUMNREDUCE`	Makes a denominator polynomial matrix column reduced
`ROWREDUCE`	Makes a denominator polynomial matrix row reduced
`RMFD`	Right Matrix Fraction Decomposition (MFD) of a transfer matrix
`LMFD`	Left MFD of a transfer matrix
`RIGHTCOPRIME`	Gives a right coprime MFD from a noncoprime MFD
`LEFTCOPRIME`	Gives a left coprime MFD from a noncoprime MFD

SS2TF	State space to transfer function conversion
MAKESYS	Makes a list of the A, B, C, D matrices to represent a system

Multivariable Realizations

CONTROLLER	Calculates a controller form realization
OBSERVER	Calculates an observer form realization
CONTROLLABILITY	Calculates a controllability form realization
OBSERVABILITY	Calculates an observability form realization

Generation of the $\mathcal{S}(A, B, C, D)$-file

TOMIMO	Generates the file ABCD.MIM from A, B, C, and D.

Example—Polynomial Matrix Manipulations

The following example is a MACSYMA *Demo* that illustrate the use of polynomial matrices for analysis of multivariable linear systems. The cumbersome manipulations are done by the above functions. The Demo starts with a transfer function matrix, describing a multivariable linear system. The description is transfered into a matrix fraction decomposition, MFD, i.e. a polynomial matrix description. Extraction of different polynomial matrix factors of the MFD are made. Also, different multivariable realizations are presented. For terminology and a background the reader is refered to [Kailath], especially Chapter 6.

The file shown is a MACSYMA log file, output with the **typeset** switch **true**. The resulting Troff/EQN typesetting code has automatically translated to TEX by the program MacEQ2TEX (see [Mårtensson, 1986]).

```
(c1) load("login.mac")$
(c2) demo("realizations.dem");
/* This demo describes a couple of examples in Kailath, chapter 6.
Example 6.2-1. Alternative MFDs for a Transfer Function. p. 368-9.
Example 6.4-1. Controller-Form Realization of a Right MFD. p. 407-8.
Example 6.4-2. Observer-Form Realization of a Left MFD. p. 416
Example 6.4-6. Constructing Canonical Controllability Forms. p. 433-4. */
(c3) g:matrix([s/((s+1)*(s+2))^2,s/(s+2)^2],[-s/(s+2)^2,-s/(s+2)^2]);
```

(d3)
$$\begin{bmatrix} \frac{s}{(s+1)^2(s+2)^2} & \frac{s}{(s+2)^2} \\ -\frac{s}{(s+2)^2} & -\frac{s}{(s+2)^2} \end{bmatrix}$$

```
/* Example 6.2-1. Alternative MFDs for a Transfer Function. p. 368-9. */
(c4) rmfd(g);
```

(d4)
$$\left[dr = \begin{bmatrix} (s+1)^2(s+2)^2 & 0 \\ 0 & (s+2)^2 \end{bmatrix}, nr = \begin{bmatrix} s & s \\ -s(s+1)^2 & -s \end{bmatrix}\right]$$

```
(c5) ev(rightcoprime(dr,nr),%);
```

(d5)
$$\left[dr=\begin{bmatrix}(s+1)^2(s+2)^2 & -(s+1)^2(s+2)\\ 0 & s+2\end{bmatrix},nr=\begin{bmatrix}s & 0\\ -s(s+1)^2 & s^2\end{bmatrix},rr=\begin{bmatrix}1 & 1\\ 0 & s+2\end{bmatrix}\right]$$

```
(c6) ev(columnreduce(dr,nr),%);
```

(d6)
$$\left[dr=\begin{bmatrix}0 & -(s+1)^2(s+2)\\ (s+2)^2 & s+2\end{bmatrix},nr=\begin{bmatrix}s & 0\\ -s & s^2\end{bmatrix},u=\begin{bmatrix}1 & 0\\ s+2 & 1\end{bmatrix}\right]$$

```
/* Example 6.4-1. Controller-Form Realization of a Right MFD. p. 407-8. */
(c7) ev(real:controller(dr,nr),%);
```

(d7)
$$\left[a=\begin{bmatrix}-4 & -4 & 0 & -1 & -2\\ 1 & 0 & 0 & 0 & 0\\ 0 & 0 & -4 & -5 & -2\\ 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1 & 0\end{bmatrix},b=\begin{bmatrix}0 & 1\\ 0 & 0\\ -1 & 0\\ 0 & 0\\ 0 & 0\end{bmatrix},c=\begin{bmatrix}1 & 0 & 0 & 0 & 0\\ -1 & 0 & 1 & 0 & 0\end{bmatrix}\right]$$

```
/* Example 6.4-6. Constructing Canonical Controllability Forms. p. 433-4.
*/
/* Search by Crate 2 */
(c8) ev(controllability(a,b,c),%);
```

(d8)
$$\left[a=\begin{bmatrix}0 & 0 & -2 & 0 & 0\\ 1 & 0 & -5 & 0 & 0\\ 0 & 1 & -4 & 0 & 0\\ 0 & 0 & 2 & 0 & -4\\ 0 & 0 & 1 & 1 & -4\end{bmatrix},b=\begin{bmatrix}1 & 0\\ 0 & 0\\ 0 & 0\\ 0 & 1\\ 0 & 0\end{bmatrix},c=\begin{bmatrix}0 & 0 & 1 & 1 & -4\\ -1 & 4 & -12 & -1 & 4\end{bmatrix}\right]$$

```
/* Search by Crate 1 */
(c9) crate_nr:1;
```

(d9) $$1$$

```
(c10) ev(controllability(a,b,c),real);
```

(d10)
$$\left[a=\begin{bmatrix}0 & 0 & 0 & -4 & 2\\ 1 & 0 & 0 & -12 & 5\\ 0 & 1 & 0 & -13 & 4\\ 0 & 0 & 1 & -6 & 1\\ 0 & 0 & 0 & 0 & -2\end{bmatrix},b=\begin{bmatrix}1 & 0\\ 0 & 0\\ 0 & 0\\ 0 & 0\\ 0 & 1\end{bmatrix},c=\begin{bmatrix}0 & 0 & 1 & -6 & 1\\ -1 & 4 & -12 & 32 & -1\end{bmatrix}\right]$$

```
/* Example 6.4-2. Observer-Form Realization of a Left MFD. p. 416 */
(c11) lmfd(g);
```

(d11) $$\left[dl=\begin{bmatrix}(s+1)^2(s+2)^2 & 0\\ 0 & (s+2)^2\end{bmatrix}, nl=\begin{bmatrix}s & s(s+1)^2\\ -s & -s\end{bmatrix}\right]$$

```
(c12) ev(leftcoprime(dl,nl),%);
```

(d12) $$\left[dl=\begin{bmatrix}(s+1)^2(s+2)^2 & 0\\ (s+1)^2(s+2) & s+2\end{bmatrix}, nl=\begin{bmatrix}s & s(s+1)^2\\ 0 & s^2\end{bmatrix}, rl=\begin{bmatrix}1 & 0\\ -1 & s+2\end{bmatrix}\right]$$

```
(c13) ev(rowreduce(dl,nl),%);
```

(d13) $$\left[dl=\begin{bmatrix}0 & -(s+2)^2\\ (s+1)^2(s+2) & s+2\end{bmatrix}, nl=\begin{bmatrix}s & s\\ 0 & s^2\end{bmatrix}, u=\begin{bmatrix}1 & -(s+2)\\ 0 & 1\end{bmatrix}\right]$$

```
(c14) ev(observer(dl,nl),%);
```

(d14) $$\left[a=\begin{bmatrix}-4 & 1 & 0 & 0 & 0\\ -4 & 0 & 0 & 0 & 0\\ 0 & 0 & -4 & 1 & 0\\ 1 & 0 & -5 & 0 & 1\\ 2 & 0 & -2 & 0 & 0\end{bmatrix}, b=\begin{bmatrix}1 & 1\\ 0 & 0\\ 0 & 1\\ 0 & 0\\ 0 & 0\end{bmatrix}, c=\begin{bmatrix}0 & 0 & 1 & 0 & 0\\ -1 & 0 & 0 & 0 & 0\end{bmatrix}\right]$$

4. Generation of a MIMO System Text File

It will now be demonstrated how a MIMO system in MACSYMA can be transfered into a text file of a special form. The special form of the text file is chosen to be the same as the print format from CTRL-C. This makes it possible to load results form MACSYMA into CTRL-C. It should also be mentioned that there is a Pascal program written by [Mårtensson, 1986] that generates Simnon code from this text file representation. The generation of the text file from MACSYMA is made by the function TOMIMO. This is a LISP program and consequently we have to enter the LISP mode before we apply it to our MIMO system.

```
(c1) load("login.mac")$
(c2) a:matrix([1,2],[3,4])$
(c3) b:matrix([5,6],[7,8])$
(c4) c:matrix([9,10],[11,12])$
(c5) d:matrix([13,14],[15,16])$
(c6) sys:makesys(a,b,c,d);
```

(d6) $$\left[a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, b = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}, c = \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix}, d = \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix}\right]$$

```
(c7)
Break Entering Lisp:
<1>: (load "tomimo.l")
t
<1>: (tomimo $sys 'abcd.mim)
t
```

The MIMO system has now been written in the file `ABCD.mim`. This file looks as follows:

```
    nmp =

    2 2 2

    a =

    1 2
    3 4

    b =

    5 6
    7 8
```

```
c =

9 10
11 12

d =

13 14
15 16
```

5. Root Locus

In this section we will demonstrate a method for computing root loci by combining symbolic and numerical computations. Only a very brief description will be given. A fuller description is given in [Holmberg, Lilja, Mårtensson, 1986].

The naive way of plotting root loci of the type

$$A(s) + kB(s) = 0$$

where A and B are polynomials and k real, is to solve the equation for a number of equi-distant k-values and then mark the roots by an '×' for each k. This method has severe disadvantages: Firstly it is rather time consuming and secondly it gives a very bad resolution near multiple roots. To be presentable, the plots also need heavy manual paste-up. In the following subsection, a method based on the implicit function theorem is suggested. A non-linear differential equation, that describes the root locus locally, is obtained by some manipulations of the transfer function (done in MACSYMA for example) and a package for solving the differential equation (e.g. Simnon) can then be utilized to compute and plot the root locus. This method is both faster and gives a better performance near multiple roots than the method mentioned above.

A general problem

This subsection proposes a method for plotting the locus of points s in the complex plane satisfying the equation

$$f(s,k) = 0, \qquad s,k \in \mathbb{C} \tag{$\flat$}$$

where f is analytic in s and k and where k is restricted to the real axis. Several common control theory problems are covered in this formulation: Ordinary root loci, LQG root loci ($k =$ control weighting), zeros of sampled systems ($k =$ the sampling interval), etc.

The method is based on the implicit function theorem applied to ($\flat$). The idea is the following: The problem is to compute $\{s | f(s,k) = 0, \quad k \in [a,b] \subset \mathbb{R}\}$. For this, compute the k's such that ($\flat$) has multiple roots in s. Away from these, the branches $s_i(k)$ satisfies

$$\frac{d}{dk}s_i(k) = -\frac{\frac{\partial f}{\partial k}}{\frac{\partial f}{\partial s}} \tag{$\sharp$}$$

Implementation

The transfer function $G(s)$ is specified in MACSYMA and the closed loop characteristic polynomial $p(s,k)$ is calculated. The real and imaginary part of the right hand side of ($\sharp$) are then computed and written to a file using the function `print_ode`. To avoid divide overflow in Simnon when a multiple root is encountered one has to stop the integration before the multiple root. For each multiple root, one therefore has to find the local behavior of s with respect to k. A graphical method to do this is to plot a

Newton diagram. This is equivalent to making the substitution $s := bk^a$ in the characteristic polynomial and then finding pairs of dominating terms. The function `newton` implements this and returns two lists. The first list contains the possible k^a-alternatives and the second list gives the corresponding coefficients (expressed as a polynomial in b). The function `near_multiple_roots` uses `newton` for calculating the values of k for which $|\frac{d}{dk}s_i(k)| = d_{max}$. The function `print_kxy` uses `near_multiple_roots` to print out these k values and the corresponding solutions in s.

The differential equation for the real and imaginary parts of the root locus is written into the file `ode.rl`. The k-values specifying the intervals for which the root locus is to be plotted for are written (together with the corresponding initial values for the branches of the root locus) into the file `rootloc.rl`. These two files are then processed by a procedure written in the "editor language" TPU (Text Processing Utility) in VMS generating one Simnon system description file `ode.t` (the "dynamics" file) and one file `rootloc.t` containing the commands for setting initial values and integrating.

An Example

The following MACSYMA dialogue shows an example where the functions `print_ode` and `print_kxy` are used. In the example the interval for the gain k is chosen to $-2 \leq k \leq 2$ and the maximum derivative to $d_{max} = 100$.

```
(c1) load("rootloc.mac")$
(c2) g:matrix([1/s^2,1/s],[-1/s^2,0]);
```

(d2) $$\begin{bmatrix} \frac{1}{s^2} & \frac{1}{s} \\ -\frac{1}{s^2} & 0 \end{bmatrix}$$

```
(c3) print_ode(g);
```

(d3) *ode.rl*

```
(c4) print_kxy(g,-2,2,100);
```

(d4) *rootloc.rl*

The resulting files `ode.rl` and `rootloc.rl` are then processed by the TPU file `rootloc.tpu` to get the Simnon system description file `ode.t` and the Simnon command file ("macro" file) `rootloc.t`. The Simnon commands required to plot the root locus are:

```
> syst ode
> axes h -2 2 v -2 2
> rootloc
```

The result is shown in Figure.

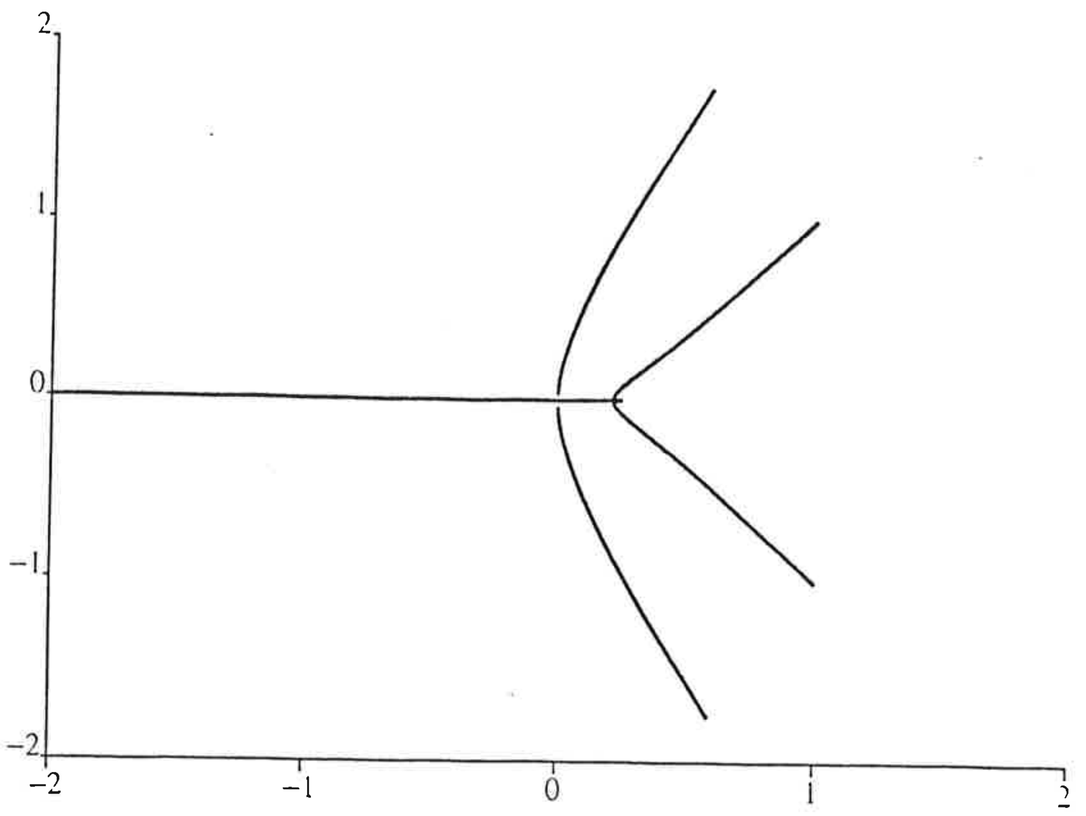


Figure: The root locus plot.

6. Conclusions

As motivated and illustrated symbolic manipulation could be very useful. The user interface could be improved by allowing the user to work on a higher level. He can present his problem on a for him suitable form. Results on analytic or symbolic form could give a better insight into structural properties than numerical tables. Even when it is not possible to carry the symbolic calculations all the way through, symbolic manipulation could be useful. Symbolic manipulation could simplify the problem and transform it to a form better suited for numerical solution. Symbolic manipulation could also be used for automatic generation of procedures for calculating gradients, Jacobians, Hessians etc. thereby relieving the user's burdon and hopefully decreasing the possibilities of introducing errors.

Our experiences of MACSYMA are that it is a powerful tool and can do a lot with proper guidance from the user. One advantage with with MACSYMA is that it is written in Lisp. This makes it possible to extend the program with Lisp functions. As you remember from Chapter 4, this enabled us to establish an interaction between MACSYMA and other programs, like Simnon and CTRL-C. The drawbacks are that it is a large program and that it consumes a lot of computer power. Unfortunately, MACSYMA is not modularized. For use in CACE systems it should be desirable to have modularized tools for symbolic calculation so that a user could select for him a proper set. We are eagerly searching for such a toolkit.

It is important to consider that there is a user in the loop. He can in many cases improve the manipulation by proposing substitutions and by informing the system on what kind of forms he want the answer. In many cases an equation system can be simplified considerably if it can be assumed that a parameter or a certain expression is zero. It is difficult for the user to anticipate all such cases in advance, but he may well be able to answer those questions interactively. Also, if the model is modified there should be facilities to take care of assumptions made before so he doesn't need to consider them once more when the manipulations are redone. To speed up the manipulation it is advisable to store the successful path and try it when the user has modified his model. The logging facility is also necessary for the explanation facility. If the numerical solution procedure fails, the error message should relate to the user's original formulation and not to the manipulated expressions.

Acknowledgements

The work has been supported by the Swedish Board of Technical Development in the project "Combination of formula manipulation and numerics" (STU–85–4809).

References

ELMQVIST, H. (1978): "A Structured Model Language for Large Continuous Systems," Ph.D-thesis TFRT-1015, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQVIST, H. and S.E. MATTSSON (1986): "A Simulator for Dynamical Systems Using Graphics and Equations for Modelling," *Proceedings of the IEEE Control Systems Society Third Symposium on Computer-Aided Control Systems Design (CACSD)*, Arlington, Virginia, September 24–26, 1986.

ELMQVIST, H., K.J.ÅSTRÖM and T. SCHÖNTHAL (1986): *Simnon—User's Guide for MS-DOS Computers*, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

HOLMBERG U. (1986): "Some MACSYMA Functions for the Analysis of Multivariable Linear Systems," Report CODEN: LUTFD2/(TFRT-7333)/1–40/(1986), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

HOLMBERG, U., LILJA, M., MÅRTENSSON, B. (1986): "Integrating Different Symbolic and Numeric Tools for Linear Algebra and Linear System Analysis," Report CODEN: LUTFD2/(TFRT-7338)/ 1–17/(1986), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

KAILATH, T. (1980): *Linear Systems*, Prentice-Hall Information and System Sciences Series, Englewood Cliffs, New Jersey.

LILJA, M. (1986): "Symbolic and Numerical Computation of Implicit Functions, Especially Root Loci," Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, to appear.

MÅRTENSSON, B. (1986): "MacEQ2TEX and S2TEX—Automatic TEX-code generation from MACSYMA and CTRL-C," Report CODEN: LUTFD2/(TFRT-7334)/1–11/(1986), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

THE MATHLAB GROUP LABORATORY FOR COMPUTER SCIENCE (1983): *MACSYMA Reference Manual*, M.I.T., Cambridge, MA.

MATTSSON, S.E. (1986): "On Differential/Algebraic Systems," Report TFRT-7327, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

MATTSSON, S.E., H. ELMQVIST and D.M. BRÜCK (1986): "New Forms of Man-Machine Interaction," Report TFRT-3181, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

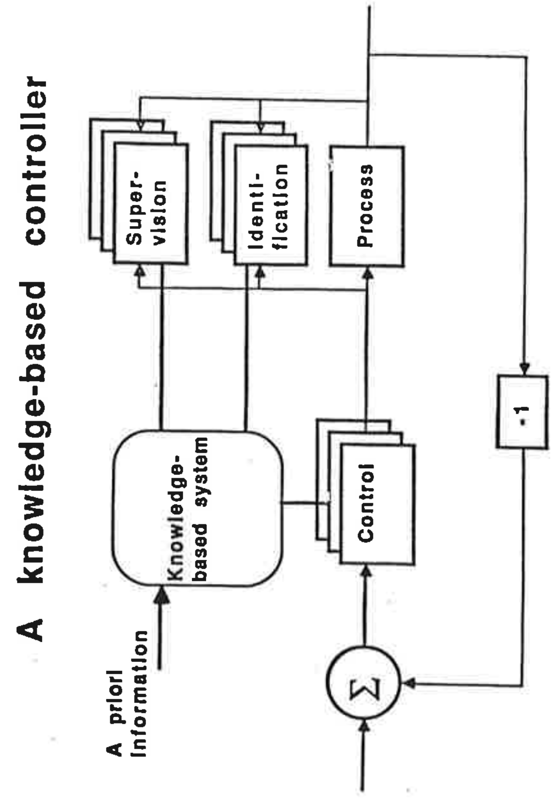
PAVELLE, R., M. ROTHSTEIN and J. FITCH (1981): "Computer Algebra," *Scientific American*, **245**, 6, 136–152.

SYSTEMS CONTROL TECHNOLOGY, INC., *CTRL-C, User's Guide*, 1801 Page Mill Road, Palo Alto, CA 94304.

Ett expertsystem för reglering på lokalnivå

Karl-Erik Årzén

Institutionen för reglerteknik
Lunds Tekniska Högskola



INDUSTRIAL PROCESS CONTROL

Many control loops are badly tuned or run in manual mode

Adaptive controllers:

Commissioning difficult

Requires prior process information

Poor operator understanding

Heuristic safety-jackets important but difficult to develop

Auto-tuners:

Limited tuning and control design methods

WHAT IS LACKING?

Diagnosis functions, loop assessment, deeper control theoretical knowledge, user query and interaction facilities

VISIONARY GOAL

A controller

- that can satisfactorily control arbitrary time-variabl non-linear processes exposed to various disturbances.
- which requires a minimal amount of prior process knowledge.
- which can make advantage of available prior process knowledge.
- that performs diagnosis of the control performance and the loop components.
- which the user can "reason" with, i.e., ask questions and get information and explanations about, e.g., process dynamics, actual control performance, achievable preformance, specification trade-offs, etc.
- where the underlying control knowledge and heuristics are transparently represented in a way that easily allows for modifications and extensions.

APPROACH

"Include an experienced control engineer in the closed control loop and provide him with the necessary toolbox of algorithms for control, identification, measurement, monitoring, and control system design."

Encode general control knowledge and heuristics about auto-tuning and adaptation in a supervisory expert system.

The controller consists of an "intelligent" combination of dedicated algorithms.

KEY PROBLEMS

What process knowledge is needed and how can it be automatically acquired?

No easy answer, several approaches

How does a suitable expert system architecture for expert control look like?

Real-time, on-line application

Knowledge represented as rules and procedures

PROSPECTS

Research areas:

Intelligent PID auto-tuners

Expert supervision of adaptive controllers

Combination of feedback control and diagnosis

Auditing of control loops

Possible short term results:

Smart single-loop controllers

Structuring of control engineering heuristics

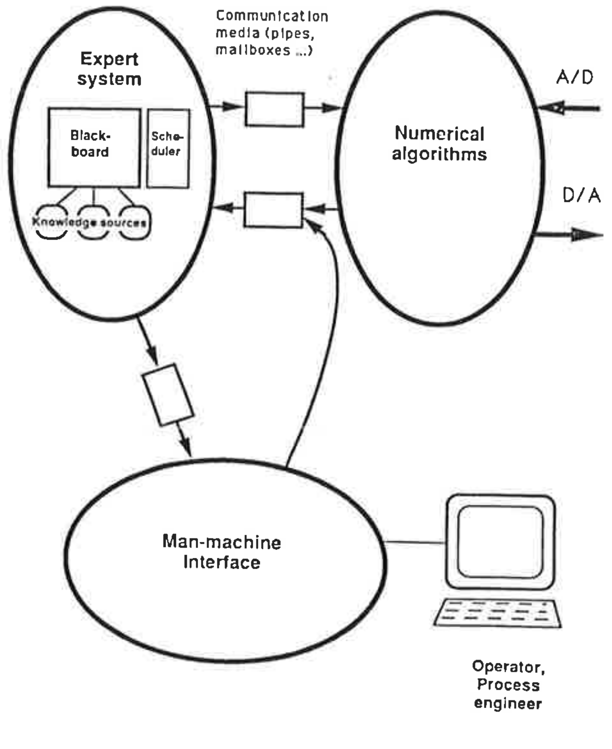
Possible long term results:

A low-level system component in the future's knowledge based control systems.

Natural extensions:

Multiloop systems

SYSTEM ARCHITECTURE



REALIZATIONS

Single computer

Distributed, multi-loop system

VAX 11/780 system:

Knowledge-based system written in LISP, YAPS, and FLAVORS.

Numerical algorithms written in Pascal.

MUSE:

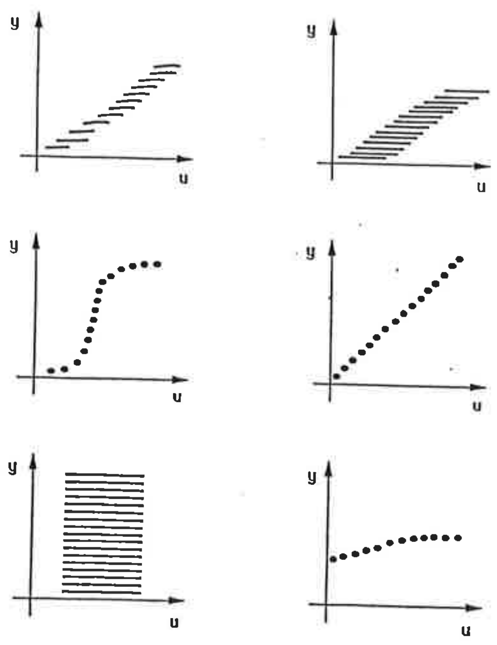
Commercial blackboard system.

Allows for emmbedded systems

Development system – Sun

STATIC CHARACTERISTICS

Can be determined from operational data



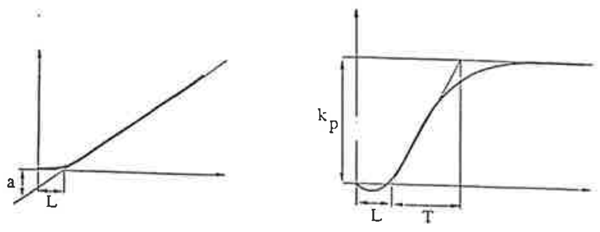
Servo or regulator (feedforward?), plant non-linearities, range and resolution of sensors and actuators

PROCESS DYNAMICS

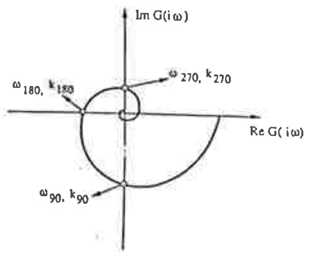
Qualitative

- stable / unstable
- monotone / oscillatory
- minimum phase / SPR / low order

Step response characteristics



Frequence response characteristics



LEVELS OF PROCESS KNOWLEDGE

L0 Qualitative classification

L1 L0 and (a,L) or (k_{180}, ω_{180})

L2 L1 and k_p

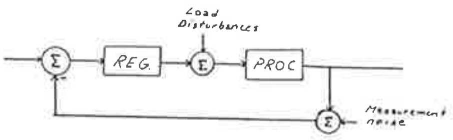
L3 L2 with more points on the step response or Nyquist curve

L4 Mathematical model with uncertainty estimates

HEURISTICS

$\omega_{PI} \approx \omega_{90}$,	$\omega_{PID} \approx \omega_{180}$
$e_{max} \approx 0.4/k_c$	PI
$e_{max} \approx 0.6/k_c$	PID
$\theta = L/T$	Rel. apparent dead-time
$\kappa = k_p/k_{180}$	Max. loop gain
$\kappa\theta \approx 1.3$	

LEVELS OF DISTURBANCE KNOWLEDGE



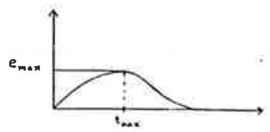
L0: Qualitative knowledge (transient,stationary, reducible,measurable,predictable)

L1: L0 + magnitude of measurement noise and load disturbances

L2: L1 + time constants associated with disturbances

L3: Matematical disturbance model with amplitude and frequency distribution

LOAD DISTURBANCES



Unit load disturbance and PID controller

$$u(t) = ke(t) + k_i \int_0^\infty e(s)ds + k_d \frac{dc}{dt}$$

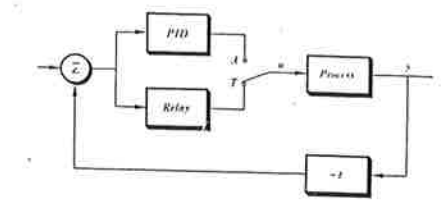
gives $\int_0^\infty e(s)ds = (u(\infty) - u(0))/k_i$

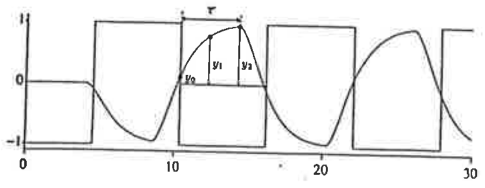
PROCESS KNOWLEDGE ACQUISITION

The relay method:

Generates an oscillation by relay feedback.

Requires the process to be essentially monotonous.





Process knowledge obtained:

Period and amplitude: $\Rightarrow k_{180}$ and ω_{180}

$y_0, y_1, y_2 \Rightarrow$ discrete model $\Rightarrow k_p, L, T$

$\tau \approx 0 \Rightarrow SPR$

Filters in the relay loop gives more points on the Nyquist curve

REGULATOR TUNING ZIEGLER-NICHOLS DESIGN

Tuning requires k_{180}, ω_{180} or a, L

Assessment requires k_p in addition

θ	Tight Control is Not Required	Tight Control is Required		
		High Measurement Noise	Low Saturation Limit	Low Measurement Noise and High Saturation Limit
Class I < 0.15	P	PI	PI or PID	P or PI
Class II 0.15 ~ 0.6	PI	PI	PI or PID	PID
Class III 0.6 ~ 1	I or PI	I + A	PI + A	PI or PID + A + C
Class IV > 1	I	I + B + C	PI + B + C	PI + B + D

A: Feedforward compensation recommended,

B: Feedforward compensation essential,

C: Dead-time compensation recommended,

D: Dead-time compensation essential.

Case II is the prime application for Ziegler-Nichols tuning

REGULATOR TUNING DOMINANT POLE DESIGN

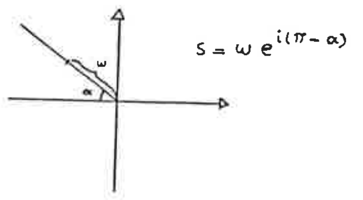
Based on the complete process model

Works with the approximation,

$$G(s) = k_p e^{-sL}/(1+sT)$$

Class of systems:

Systems with poles close to the negative real axis

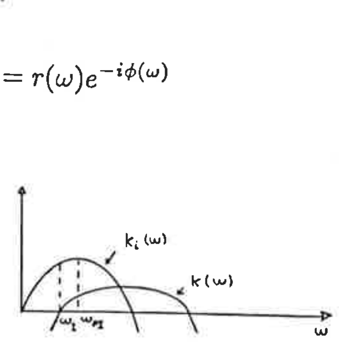


$$G(\omega e^{i(\pi-\alpha)}) = r(\omega)e^{-i\phi(\omega)}$$

The PI case

$$k(\omega) = \frac{\sin(\phi(\omega)-\alpha)}{r(\omega)\sin(\alpha)}$$

$$k_i(\omega) = \frac{\sin(\phi(\omega))}{r(\omega)\sin(\alpha)}$$



EXAMPLE 1

$$G(s) = \frac{e^{-sL}}{1+sT}$$

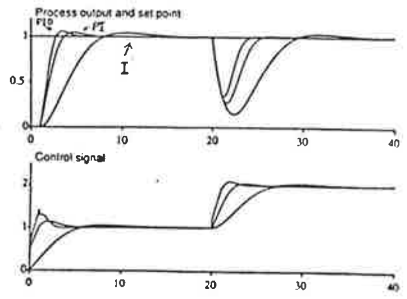
$L = T = 1$

Crude assessment

$\omega_{90} = 0.9$	$w_{180} = 2.0$
$k_{90} = 0.76$	$k_{180} = 0.44$

Accurate assessment

I:	$\omega = 0.55$ $k_i = 0.36$
PI:	$0.55 \le \omega \le 1.1$ $0.36 \le k_i \le 0.67$
PD:	
PID:	$0.9 \le \omega \le 1.7$ $k_i \le 0.76$



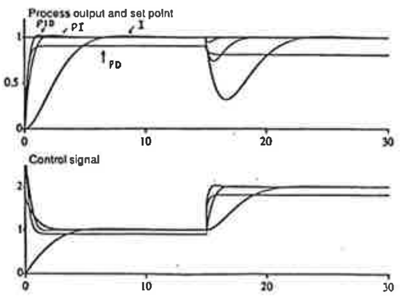
$$G(s) = \frac{1}{(1+s)(1+0.2s)(1+0.05s)(1+0.01s)}$$

Crude assessment

$\omega_{90} = 3$ $\quad w_{180} = 32$
$k_{90} = 0.3$ $\quad k_{180} = 0.03$
$\kappa = 33$

Accurate assessment

I:	$\omega = 0.62$
PI:	$0.62 \le \omega \le 2.5$ $0.4 \le k_i \le 2.4$
PD:	$3.7 \le \omega \le 11$ $1.9 \le k \le 9.7$
PID:	$2.2 \le \omega \le 7.5$ $2.0 \le k_i \le 27$

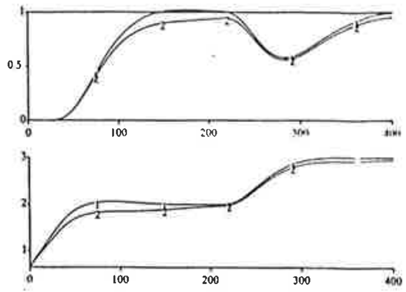


PIDWIZ – commercial control loop tuner

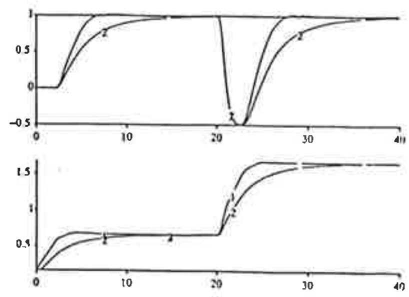
Step response and load disturbance

1: LTH-design, 2:PIDWIZ

$$G(s) = \frac{0.5e^{-10s}}{(s+1)..(10s+1)}$$



$$G(s) = \frac{1.5e^{-2s}}{(1+0.2s)^5}$$



PARTICIPATORS

Karl Johan Åström, Karl-Erik Årzén, Per Persson, Tore Hägglund – Automatic Control, LTH

C. C. Hang – Univ. of Singapore

PUBLICATIONS

ÅRZÉN, K-E. (1987): "Realization of expert system based feedback control," Ph D thesis, TFRT-1029, Department of Automatic Control, Lund.

ÅRZÉN, K-E. (1988a): "An architecture for expert system based feedback control," *Proc. IFAC Workshop on AI in real-time control*, Swansea, Wales.

ÅRZÉN, K-E. (1988b): "Expertsystem klarar svår process," *Elteknik*, 19.

ÅRZÉN, K-E. (1988c): "Remarks and suggestions concerning G2 version 1.1," Technical report TFRT-7409, Department of Automatic Control, Lund Institute of Technology, Lund.

ÅSTRÖM, K.J. (1988a): "Towards intelligent control," *Proc. American Control Conference*, Atlanta, GA.

ÅSTRÖM, K.J. (1988b): "Assessment of achievable performance of simple feedback loops," *Proc. Conf. on Decision and Control*, also as Technical report TFRT-7411.

ÅSTRÖM, K.J., C.C. HANG och P. PERSSON (1988): "Heuristics for assessment of PID control with Ziegler-Nichols tuning," Technical report TFRT-7404, Department of Automatic Control, Lund Institute of Technology, Lund.

HANG, C.C. och K.J. ÅSTRÖM (1988): "Refinements of the Ziegler-Nichols tuning formula for PID auto-tuners," *Proc. ISA Annual Conf. Houston USA*.

Using Control Theory to Improve Stepsize Selection in Numerical Integration of ODE

Kjell Gustafsson
Department of Automatic Control, Lund Institute of Technology
P.O. Box 118, S-221 00 LUND, Sweden, Email: kjell@control.lth.se

Abstract. Stepsize selection in numerical integration of ordinary differential equations can be regarded as a control problem. An estimate of the solution error is fed back in order to choose the new stepsize. The standard stepsize controller used today does not give satisfactory performance. It is based on a static asymptotic relation between the stepsize and the error. A dynamic model that better describes this relation is derived. It is then used to analyze and to explain the problems with the standard controller. A new controller is designed using the model. It is of PI type and gives superior performance at little extra expense.

Keywords: Control applications, computer simulation, numerical analysis, numerical integration, stepsize selection, ordinary differential equations

1. Introduction

When solving ordinary differential equations numerically (e.g. simulating the time response of a continuous time control system) it is often hard to judge the quality of the produced solution. Simulation programs are normally constructed such that the user only supplies the differential equation and an accuracy requirement, while the program takes care of everything regarding the numerical integration. The user trusts the program to efficiently produce a solution within the required accuracy. Normally the program succeeds in doing this, but for some, remarkably simple, equations even the best algorithms known today fail. An example from a simulation of a control system is shown in Fig. 1. The oscillating component in the signal to the left is not a part of the true solution of the equations. The simulation program does not succeed in detecting this, even though the errors are much larger than the accuracy requirement.

The artifact in Fig. 1 is caused by the algorithm for stepsize control in the integration method. Consider the initial value problem

$$\dot{y} = f(t, y), \quad 0 \le t \le T, \quad y, f \in \mathcal{R}^p \qquad y(0) = y_0 \tag{1}$$

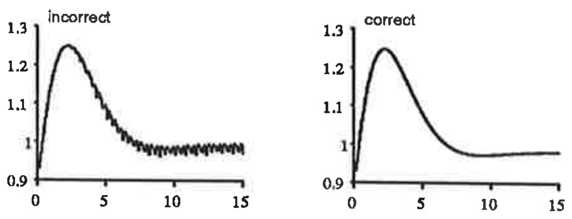


Figure 1. A simulation of the control signal during a step response of a simple control system. The oscillatory component, in the signal to the left, is caused by an irregular stepsize sequence. The correct signal, to the right, is obtained by improving the stepsize control algorithm.

with the exact solution $y(t)$. An integration method forms the numerical solution $\{y_n\}_{n=0}^{N}$ at discrete time points t_n using a discretization of (1). The (time)distance between two solution points is called the stepsize h_n, $h_n = t_{n+1} - t_n$. It controls the quality of the numerical solution, and the goal is to choose it such that the difference between $y(t_n)$ and y_n is small.

The appropriate stepsize h_n varies along the solution of the differential equation. It is hard for the user to relate a given stepsize to a specific accuracy and therefore the choice is normally left to the simulation program. The choice is a matter of both accuracy and efficiency.

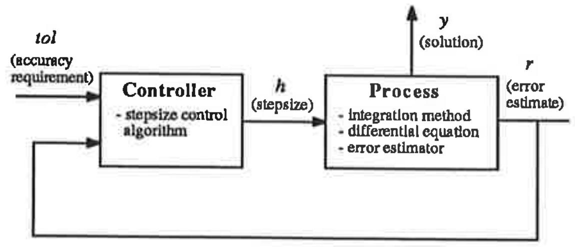


Figure 2. Control system view of stepsize selection.

Small steps make the solution accurate, but require more computation due to the increase in the number of steps needed. Therefore, the strategy is to choose the stepsize as large as possible within the accuracy requirements, since that gives an acceptable solution with the least amount of computation.

The stepsize selection can be regarded as a control problem (see Fig. 2). An estimate of the solution error is compared with the user-specified accuracy requirement, and the result is fed back and used to determine the new stepsize. The controller should keep r close to *tol*, and when doing so preferably use a smooth control signal. A smooth stepsize sequence improves the quality of the error estimate, and then it is more likely that r really reflects the difference between y_n and $y(t_n)$.

The standard rule for stepsize selection is derived from a static asymptotic relation between the stepsize and the estimated error. In reality the relation is dynamic and depends on the operating conditions. Often the static relation is a good approximation, but in some cases a dynamic model has to be used. The standard controller can not handle these dynamics, and the result is an unstable control system manifested as in Fig 1. The problems in Fig 1 is easily detected, but often the discrepancies are more subtle, and for the user hard to detect.

In this paper we will derive a model that captures the process behavior. The model is then used to derive a new controller, which solves the stepsize control problem. Finally, the properties of the new controller are demonstrated using a numerical example.

2. Standard Stepsize Control

To gain insight and to introduce notations, we will start by deriving the standard stepsize controller. As will be seen it can be viewed as a pure integrating controller, which explains some of its properties.

An explicit m-stage Runge-Kutta method (Hairer, Nørsett, and Wanner, 1987) applied to the initial-value problem (1) takes the following form

$$\begin{aligned}
\dot{Y}_i &= f(t_n + c_i h_n, y_n + h_n \sum_{j=1}^{i-1} a_{ij}\dot{Y}_j), \\
&\quad i = 1 \ldots m \\
y_{n+1} &= y_n + h_n \sum_{j=1}^{m} b_j \dot{Y}_j \\
t_{n+1} &= t_n + h_n \\
\hat{e}_{n+1} &= h_n \sum_{j=1}^{m} (b_j - \hat{b}_j)\dot{Y}_j \\
r_{n+1} &= \|\hat{e}_{n+1}\|
\end{aligned} \tag{2}$$

The coefficients $\{a_{ij}\}$, $\{b_i\}$, $\{\hat{b}_i\}$, and $\{c_i\}$ are chosen such that the Taylor expansion of the numeric solution y_n matches as many terms as possible of the Taylor expansion of the true solution $y(t_n)$. The exponent of the matched term with the highest order is referred to as the method order. The method supports two formulae of orders $k-1$ and k, respectively. They are represented by the two coefficient sets $\{b_j\}$ and $\{\hat{b}_j\}$. One coefficient set is used to advance the integration while the other is used for the error estimate $\hat{e}$.

For robustness, a mixed absolute-relative "norm" is used to form r. A common choice is

$$\|\hat{e}\| = \sqrt{\sum_i \left(\frac{\hat{e}_i}{|y_i| + \eta_i}\right)^2} \tag{3}$$

where η_i is a scaling factor for the i:th component of y.

The nonlinear difference equations in (2) constitute the true process. Its behavior is complicated, but the main properties can normally be captured using a simple linear model.

The error $\hat{e}$ is formed using the difference between the two coefficient sets $\{b_j\}$ and $\{\hat{b}_j\}$, and asymptotically it is proportional to h^k, i.e

$$r_{n+1} = \|\phi_n\| h_n^k. \tag{4}$$

The coefficient vector ϕ consists of elementary differentials of f of order $k-1$ and higher (Gustafsson, 1988), and is $O(1)$ as $h \to 0$. The standard stepsize control algorithm is based on assuming ϕ constant or slowly varying. The control objective is to make r equal *tol*, and that may be achieved in the next step by choosing $h_n = tol/\|\phi_n\|$. Here ϕ_n is unknown, but since ϕ is constant, ϕ_{n-1} can be calculated from r_n and h_{n-1} and used instead. Then

$$h_n = \left(\frac{tol}{r_n}\right)^{1/k} h_{n-1} \tag{5}$$

Augmented with some safety factors, this is the stepsize controller found in most production codes used today (Gear, 1971; Hairer, Nørsett, and Wanner, 1987).

Using logarithms makes (5) read

$$\log h_n = \log h_{n-1} + k^{-1}(\log tol - \log r_n), \quad (6)$$

which can be recognized as an integrating controller with $\log tol$ as set point and $1/k$ as integration gain. Since the process model (4) can be written

$$\log r_n = k \log h_{n-1} + \log \|\phi\|, \quad (7)$$

the closed loop from $\log tol$ to $\log r_n$ will be of first order. Choosing the integration gain as $1/k$ makes it deadbeat. Moreover, the constant load disturbance $\log \|\phi\|$ is eliminated by the integrator in the controller.

Although assumed constant, the disturbance ϕ varies along the solution of the differential equation. Sometimes the variations are very large, and the controller does not manage to keep $r \approx tol$. A large increase in ϕ will result in a similar increase in r. An error r substantially larger than tol cannot be tolerated. The step has to be rejected, and a new try is made with a smaller stepsize. At times when ϕ increases drastically there may be long sequences of alternating rejected and accepted steps.

A more severe problem is that the model (4) is not adequate. The process changes behavior between different operating conditions. The controller (5) fails to handle this changing behavior, and the stepsize control loop becomes unstable, causing a highly irregular stepsize sequence. The phenomenon has been studied before, but almost all studies have focused on characterizing and describing the behavior (Shampine, 1975; Hall, 1985, 1986; Hall and Higham, 1987). Here the goal is instead to solve the problem, and in order to do that a better process model is needed.

3. A Process Model

Consider the test problem

$$\begin{aligned} \dot{y} &= \lambda y \qquad t \geq 0, \quad \lambda < 0 \\ y(0) &= y_0. \end{aligned} \quad (8)$$

Applied to (8), the Runge-Kutta algorithm can be expressed as the exact process

$$\begin{aligned} y_{n+1} &= P(h_n\lambda) y_n \\ \hat{e}_{n+1} &= E(h_n\lambda) y_n \end{aligned} \quad (9)$$

where $P(h_n\lambda)$ and $E(h_n\lambda)$ are polynomials in $h_n\lambda$.

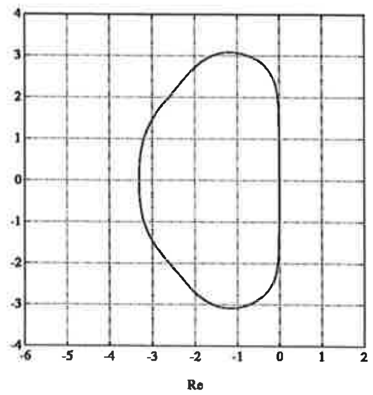


Figure 3. Stability region for DOPRI45 with local extrapolation.

EXAMPLE 1—DOPRI45
DOPRI45 (Hairer, Nørsett, and Wanner, 1987) is a fifth order explicit Runge-Kutta method. Its error estimator is also of fifth order, and

$$\begin{aligned} P(z) &= 1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24} + \frac{z^5}{120} + \frac{z^6}{600} \\ E(z) &= -\frac{97z^5}{120000} + \frac{13z^6}{40000} - \frac{z^7}{24000}. \end{aligned}$$

Up to and including the fifth order term, $P(z)$ is a correct Taylor expansion of e^z (the solution of (8). The region defined by $\mathcal{S} = \{h\lambda : |P(h\lambda)| \leq 1\}$ (see Fig. 3) is referred to as the stability region of the method. □

When $h_n\lambda$ is small and well inside the stability region $\mathcal{S}$ of the Runge-Kutta method, the process is well described by the process model (4). To verify, observe that $E(h_n\lambda)$ takes the form $E(h_n\lambda) = \kappa_0(h_n\lambda)^k + \kappa_1(h_n\lambda)^{k+1} + \ldots$, and hence the error estimate

$$\begin{aligned} r_{n+1} &= \|\phi_n\| h_n^k, \\ \phi_n &= y_n \lambda^k (\kappa_0 + \kappa_1 h_n \lambda + \ldots). \end{aligned} \quad (10)$$

Here ϕ_n is measured with the same norm as $\hat{e}$. The coefficient vector ϕ_n is varying along the solution y_n. It is also dependent on h_n, but the dependence is weak since $|\kappa_0| \gg |\kappa_1 h_n \lambda + \ldots|$ when $h_n\lambda$ is small, which is the case when $h_n\lambda$ is well inside $\mathcal{S}$.

For the linear problem (8), $\phi \to 0$ as $t \to \infty$, and to keep r equal to tol, the stepsize controller will increase the stepsize. As h_n increases, it will eventually equal h_s where h_s puts $h_s\lambda$ on $\partial\mathcal{S}$, the border of the stability region $\mathcal{S}$ of the integration method (for DOPRI45 $h_s\lambda = -3.31$, see Fig. 3). Further increasing the stepsize makes the nonlinear difference equation system (9) unstable, and the stepsize is said to be limited by numerical stability. The behavior of (9) changes when $h_n\lambda$ approaches ∂S, and the process model (4) no longer holds. Instead a new model has to be derived.

The constant stepsize h_s leads to the stationary solution $|y_{n+1}| = |y_n|$, since $|P(h_s\lambda)| = 1$. Consider small perturbations, i.e. $h_n \approx h_s$. Then the process may be written (Gustafsson, 1988)

$$r_{n+1} = \left(\frac{h_n}{h_s}\right)^{C_1}\left(\frac{h_{n-1}}{h_s}\right)^{-C_1+C_2} r_n \tag{11}$$

with

$$\begin{aligned} C_1(h_s\lambda) &= h_s\lambda\frac{E'(h_s\lambda)}{E(h_s\lambda)}, \\ C_2(h_s\lambda) &= h_s\lambda\frac{P'(h_s\lambda)}{P(h_s\lambda)}. \end{aligned} \tag{12}$$

Again using logarithms, the process model reads

$$\begin{aligned} \log r_n &= G_p(q)\left(\log h_n - \log h_s\right), \\ G_p(q) &= \frac{C_1 q + C_2 - C_1}{q(q-1)} \end{aligned} \tag{13}$$

where q is the shift operator.

Although (13) was derived for the simple test problem (8) it is valid also for other differential equations. In (Gustafsson, 1988) the result is generalized to general linear differential equations, and also experimentally verified for nonlinear f.

4. A New Controller

The standard integrating controller (5) gives unsatisfactorily performance for the process model (13). For many commonly used Runge-Kutta methods, e.g. RKF23, RKF45, DOPRI45, the values of C_1 and C_2 are such that the closed loop system is unstable or close to instability (Hall and Higham, 1987).

In numerical analysis one does not normally regard the stepsize control system as dynamic. The relation (4) is assumed static with ϕ constant, and any discrepancy between r and tol should (and could) be fully compensated for in the next step. Therefore, the stepsize selection rule (5) is never questioned, and all attempts to solve the problem have concentrated on the process. This shows up in (Hall and Higham, 1987) where the process (the integration method) is modified to make the closed loop stable. When constructing an explicit Runge-Kutta method there is some freedom in the choice of parameters. Normally this freedom is used to minimize error coefficients or to maximize the stability region of the method, but Higham and Hall exploit it to change C_1 and C_2. Hence, the numerical properties of the integration method are traded for a stable closed loop system. From a control theory point of view, it seems more appropriate to change the controller, and reserve the free parameters to optimize the numerical properties of the integration method.

The operating conditions giving rise to the model (4) are by far the most common. Therefore, the new controller should give good performance for the case (13) without sacrificing performance for (4). For the values of C_1 and C_2 appearing in practice it is sufficient to replace the standard controller by a controller of PI type, i.e.

$$G_c(q) = k_I\frac{q}{q-1} + k_P \tag{14}$$

to achieve this end.

It is hard to give general formulae for how to choose the controller parameters k_I and k_P. Their values are a compromise between stability and response time. The coefficients C_1 and C_2 vary for different integration methods, and one cannot expect to find values that will be acceptable for all integration methods. Still, for a given method it is quite straight forward to determine controller parameters using methods like root locus plots and Nyquist plots. A fairly detailed description of the derivation of parameters for DOPRI45 is included in (Gustafsson, 1988). For DOPRI45 the parameters where chosen as $k_I = 0.06$ and $k_P = 0.13$, which should be compared with the integration gain $1/k = 0.2$ (the error estimator in DOPRI45 is of order 5) in the standard controller.

The new PI-controller both stabilizes (13), and improves the performance for the normal process model (4). The closed loop is no longer deadbeat, and as a result the control signal (the stepsize sequence) is smoother. This is an important improvement, since a smooth stepsize sequence makes the error estimate better, and consequently r will more accurately reflect the difference between $y(t_n)$ and y_n.

The PI-controller can be written on a form resembling the standard controller (5). Some manipulations applied to $\log h_n = G_c(q)(\log tol - \log r_n)$ using (14) yield

$$h_n = \left(\frac{tol}{r_n}\right)^{k_I}\underbrace{\left(\frac{r_{n-1}}{r_n}\right)^{k_P}}_{\text{new factor}} h_{n-1} \tag{15}$$

From this expression it is clear that the new factor corresponds to taking the most recent development of r into account when deciding upon the next stepsize. It is also clear that this type of controller is trivial to implement in existing ODE codes.

Rejected Steps

Also with the new controller, r will occasionally be too large, and the step has to be rejected. The closed loop is no longer unstable for the case (13), and the most likely reason for the large error is an increase in ϕ. After the rejected step, the next step is a retry and from the previous attempt it is known what to expect ahead. The disturbance ϕ can be calculated from the rejected step and the value is then used to determine a new stepsize. The formula is identical to the one in the standard controller (5).

It does not suffice to determine a new stepsize to restart the controller. Also its internal state has to be updated. The disturbance ϕ exhibits a lot of structure, and since it increased in the last step it is likely that it will increase in the next step too. In order not to get another rejected step, the stepsize should be decreased in the next step as well. The internal controller state can be used to achieve this end. In other words, after a rejected step, a new stepsize is calculated from (5). If it leads to an accepted step, the controller state is updated such that if the accepted step is perfect ($r = tol$), there will still be a stepsize decrease of the same factor as the one in the last step. If, on the other hand, the step is rejected, (5) is used again.

In a way this strategy for rejected steps is an ad hoc solution. Still, it works very well in practice (Gustafsson, 1988), and it has the advantage of being easy to implement.

To summarize this section, an outline of the code needed to implement the new controller is presented in Listing 1. The controller is called after each step in the integration routine, and it calculates the stepsize to be used in the next step. The variable x is the controller state, and as before, h is the stepsize and r the corresponding error estimate.

5. Numerical Test

To demonstrate some of the properties of both the old standard controller as well as the new one, they will be used to simulate the step response of a small control system (A more extensive set of numerical tests can be found in (Gustafsson, 1988) and (Gustafsson and coworkers, 1988)). The integration method is DOPRI45 with local extrapolation.

The control system is the one used in Fig. 1, and consists of a standard PID-controller with

```
if current_step_accepeted then
    if previous_step_rejected then
        $x := h \cdot h/x$
    endif
    $x := \left(\frac{tol}{r}\right)^{k_I} \left(\frac{oldr}{r}\right)^{k_P} x$
    $h := x$
    $oldr := r$
else
    $h := \left(\frac{tol}{r}\right)^{1/k} h$
endif
```

Listing 1. An outline of the code needed to implement the new controller including the restart strategy after rejected steps.

filtered D-part

$$G_{PID} = K\left(1 + \frac{1}{T_i s} + \frac{sT_d}{sT_d/N + 1}\right),$$

and the plant $1/(s+1)^4$. The parameters $K = 0.87$, $T_i = 2.7$, $T_d = 0.69$, and $N = 30$ yields a well tuned controller.

Figure 4 shows some signals originating from a simulation of the step response of the system. The figure consists of six small plots, with all signals plotted as function of time. The upper left plot shows a correct simulation (using the new stepsize controller) of the control signal (u) and the plant output (y). The upper right plot shows two curves corresponding to the work needed to solve the problem. It is the total number of integration routine calls for both the old standard controller (solid line) and the new controller (dashed line). Rejected steps are also included to properly reflect the total work. The two plots in the middle show the estimated error r (normalized with tol) for the old (left) and the new (right) controller. The two lower plots compare the stepsizes used by the two controllers.

The PID-controller and the plant form a sixth order system with four complex eigenvalues and two real. Five of the eigenvalues have a magnitude approximately equal to 1, while the sixth eigenvalue $\lambda_6 \approx -40$. The eigenvalue λ_6 is related to the filtering of the D-part in the controller.

When simulating the step response the transient corresponding to λ_6 dies out very fast. Consequently the stepsize controller increases the stepsize and soon $h\lambda_6$ is placed on ∂S. For the standard controller this leads to instability, resulting in an irregular stepsize sequence which

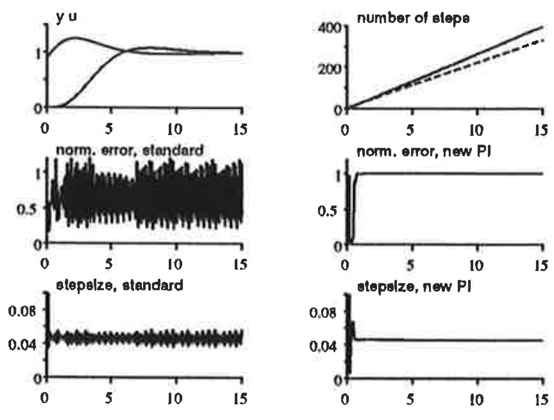


Figure 4. Simulation with $tol = 10^{-2}$.

excites the fast mode corresponding to λ_6. The error estimator fails to recover this mode properly and the produced solution is erroneous (see Fig. 1).

In contrast to the standard controller, the new controller quickly finds the correct stepsize and manages to control the error almost perfectly. Moreover, the new controller drastically decreases the number of rejected steps, and the step response is simulated with 20 % less work.

6. Conclusions

Over the years automatic control has benefited by the progress in numerical analysis. This paper has presented a problem where instead control theory is used to understand and solve a problem in numerical analysis. Viewing stepsize selection as a control problem separates an integration routine into two parts: the process (integration method, differential equation and error estimator) and the stepsize controller. Hence an integration method can be constructed for optimal numerical behavior, and then a fitting stepsize controller is designed.

To design the stepsize controller a process model is needed. Normally, a static asymptotic relation between the stepsize and the estimated error is assumed, but the relation is better described by a dynamic model when numerical stability limits the stepsize. Such a model was derived for explicit Runge-Kutta methods.

Using the dynamic model, it is straightforward to analyze the standard stepsize controller. The analysis gives insight and clearly points out that there are operating conditions where the standard stepsize controller fails to stabilize the process.

The standard stepsize controller can be recognized as a pure integrating controller. The generalization to a PI-controller is then natural, and using root loci plots or similar techniques its parameters can be tuned such that good control is achieved. The new PI-controller gives better overall performance at little extra expense.

Here only explicit Runge-Kutta methods were considered, but the same problems show up also for other types of integration methods. Similar analytical models can probably be derived for these methods as well, and once a model is obtained it can be used to analyze and improve the stepsize control.

Acknowledgements

The work has been supported by the Swedish National Board for Technical Development (STU) under contract 88-01611P.

7. References

Gear, C. W. (1971): *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall.

Gustafsson, K., Lundh, M., and Söderlind, G. (1988): "A PI stepsize control for the numerical solution of ordinary Differential Equations," *BIT*, **28**, 2, 270–287.

Gustafsson, K. (1988): *Stepsize Control in ODE-Solvers–Analysis and Synthesis*, Licentiate Thesis TFRT-3199, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Hairer, E, S. P. Nørsett, and G. Wanner (1987): *Solving Ordinary Differential Equations, I*, Springer.

Hall, G. (1985): "Equilibrium states of Runge-Kutta schemes: part I," *ACM Transactions on Mathematical Software*, **11**, 3, 289–301.

Hall, G. (1986): "Equilibrium states of Runge-Kutta schemes: part II," *ACM Transactions on Mathematical Software*, **12**, 3, 183–192.

Hall, G., and Higham, D. J. (1987): "Analysis of stepsize selection for Runge-Kutta codes," NA report No. 137, University of Manchester.

Higham, D. J., and Hall, G. (1987): "Embedded Runge-Kutta formulae with stable equilibrium states," NA report No. 140, University of Manchester.

Shampine, L. F. (1975): "Stiffness and nonstiff differential equation solvers," in L. Collatz (Ed.): *Numerische Behandlung von Differentialgleichungen*, Information Series of Numerical Mathematics 27, Birkhauser Verlag, Basel, pp. 287–301.

Särtryck "Proceedings of the USENIX C++ Technical Conference"
Denver, Colorado, October 17-20, 1988, pp. 183-192.

Modelling of Control Systems with C++ and PHIGS

Dag M. Brück

Department of Automatic Control
Lund Institute of Technology
Box 118, S-221 00 Lund, SWEDEN

E-mail: dag@control.lth.se

Abstract

This paper describes an interactive tool for modelling of control systems. The focus is on practical experiences with C++ as a development tool, and the need for multiple inheritance, parameterized types, and exception handling, in this application. Experiences with a new graphics standard, PHIGS, using an object-oriented programming style, are briefly covered.

1. Introduction

Modelling has traditionally been one of the main topics in control engineering. Control systems are complex and require careful design and analysis, in particular, as errors in control system design can become expensive. There exists today a great need for computer aided design of control systems.

Our research is centered around tools for model development and simulation. The objective is to design the basic concepts needed for structuring models, and to design the internal computer representation of control system models. An experimental tool for modelling and simulation has been developed in KEE, an expert system shell.

The experimental tool will form the basis of an engineering tool for the designer of control systems. In such a product, flexible, efficient and affordable system software must be used. We have therefore evaluated C++ as the future implementation language, and PHIGS as the main graphics system. A simplified experimental tool has been implemented in C++. Whereas the KEE version supports all essential parts of an engineering tool, the C++ version only provides graphical interaction; the internal structure is quite similar, in order to meet future needs.

2. Modelling of control systems

The model of a control system can be regarded as a hierarchy of components. One of the fundamental ideas is to build libraries of component models, ranging from basic items (for example, a pump) to more complex objects (for example, a distillation column). The designer has the option of working bottom-up, putting predefined components together to form a new component, or top-down, decomposing a complex object into manageable pieces, or most likely, a combination of bottom-up and top-down design [Nilsson, 1987]. The key word is reuseability — of earlier designs and of standard components.

A single component can be described in many ways: graphically, textually, using block diagrams (describing its structure), or mathematically (for example, in state-space or transfer-function form). It is also necessary to use models with different degrees of detail and complexity, for example, an efficient simulation model for normal operation, and an extended model for analyzing error conditions. All these models are needed in different stages of the design, and should be available in a model development tool. It should be noted that the common "machine" view may be replaced by a "materials" view. For example, a chemical compound may carry all knowledge in the model, while the stations in the refinery only signal changes of state.

With our set of basic concepts, a model has three properties: it has terminals which provide an interface to the outside world, parameters for adapting its behaviour, and at least one realization that defines its behaviour. Only data in the terminals are available to other components; there are no global data, except a time reference for simulation.

We currently support two types of realizations: primitive realizations using ordinary differential equations, and structured realizations using block diagrams. A structured realization consists of submodels and connections (between submodels, and between submodels and the terminals of the enclosing model). Interaction between components is defined only by connections.

Simulation is often used to analyze control systems, and the designer should be able to simulate his/her model using this tool. Simulation introduces a number of interesting mathematical problems, which will not be covered further in this paper [Mattsson, 1988b]. The connection concept also raises interesting questions: for example, what is a legal connection, and how do you define compatibility between terminals [Mattsson, 1988a].

According to current trends, it is also necessary to throw in an expert system and a couple of knowledge bases.

3. Direct model representation

Modelling of control systems maps nicely to the ideas in object-oriented programming. It is natural to represent a model with a class in the programming language used for implementing the design tool. It is then possible to develop new models using inheritance and specialization of classes.

Inheritance is not suitable for describing all kinds of relationships between models. Multiple representations of a single model (textual or mathematical), and specialization (a car is a special kind of vehicle), can be described with inheritance. Decomposition of a model into its components is different. For example, that a car has tyres does not mean that the car can be inflated, so inheritance is not the right mechanism; components are represented by class members (Listing 1).

The direct way of representing models with classes is used in the experimental tool developed in KEE. Instantiation is used, for example, to create objects that contain simulation data. A necessary key feature of KEE (and object-oriented systems like Loops) is the possibility to dynamically define new classes while the program is running.

```
class vehicle {
  char*  owner;
};

class car : public vehicle {
  tyre   fl, fr, rl, rr;
  engine e;
};
```

Listing 1. Direct representation of a car model, derived from vehicle.

4. Model representation in C++

If interactive model development is presumed, direct representation is not possible in C++, simply because classes cannot be defined at runtime. Consequently, components cannot be represented directly with class members, and inheritance cannot be used to derive new models. To be able to interactively create models, we must implement a dynamic framework for representing models, realizations, etc. This framework is similar to the class systems commonly based on Lisp, but the implementation task is simplified by the structure of control systems.

It should be noted that the engineer developing control systems will see an interactive modelling tool; C++ is used only to implement the dynamic framework, not as a control system description language. One can also say that the object-oriented aspects of model representation have been separated from the object-oriented aspects of C++. Still, object-oriented programming effectively supports the design and implementation of the framework.

Internal data structures

Now, let's plunge straight into the internal data structures of the C++ program. The code listed below is slightly simplified; constructors and destructors are not listed, and most general purpose routines have been omitted. An example will be given below.

All objects are components; they have a name, and they can be inserted into lists (Listing 2).

```
class component {
  char* name;
  link  next;

public:
  virtual void menuaction();
  virtual void redraw();
};
```

Listing 2. Definition of the basic component class.

Method `redraw` is a schoolbook virtual function in C++: every component has a graphical representation, so all components must implement `redraw` in some way. Graphics will be described further in Section 5.

When the user points at a component and presses a mouse button, some components (e. g., models and realizations) will respond by displaying a menu. Other components (e. g., terminals and connections) are not associated with a menu. In C++, which in its present shape only supports single inheritance, method `menuaction` must be declared as a virtual function in the base class, component. When multiple inheritance becomes available in C++, menuaction would more naturally be the property of a class associated purely with the user interface; models and realizations would be derived from this class, but not terminals and connections [Stroustrup, 1987a].

Generally speaking, multiple inheritance enables us to separate the user interface and the modelling structure more effectively. There will be one "thread" of inheritance for the user interface (drawing block diagrams, and menu actions when applicable), and one thread of inheritance for the modelling of control systems (components, models, etc.). The development of class libraries, in particular, will benefit from multiple inheritance. For example, functions provided by the operating system and the window manager, will be easier to describe and use in an object-oriented fashion with multiple inheritance.

The model contains terminals and realizations, in C++ represented with linked lists (Listing 3). General purpose lists of components are used, which effectively corrupts the type security in C++. In addition, the programmer must bother about explicit type conversions. Alternatively, generic lists could be faked with macros. Future versions of C++ may incorporate true generics, also called parameterized types [Stroustrup, 1987b]. The need is evident, even in this small example.

```
class model : public component {
  list terminals;
  list realizations;

  void new_terminal();
  void new_realization();

public:
  void menuaction();
  void redraw();
};
```

Listing 3. Definition of the model class.

There are two different kinds of model realizations: primitive realizations based on equations, and structured realizations based on hierarchical block diagrams (Listing 5). There is no "one-of" concept (for example, allowing a pointer to a set of classes) in C++, so an additional class realization is needed (Listing 4). In this case, there are no real problems; in other cases, an awkward data structure might be forced upon the programmer. The one-of concept is available with full type checking in KEE, and has reduced the need for common base classes.

```
class realization : public component {
};
```

Listing 4. The common part of all realizations.

A submodel establishes a relation between two models, one fully enclosed in the other (Listing 6). With a structured realization, a model is described by the behaviour of its submodels and by its connections. The submodel also has a graphical meaning. When a model is simulated, the submodel must be "instantiated" by the model representation framework. Although many submodels may refer to a single model (e. g., a pump), every submodel requires a private data area to hold simulation variables.

```
class eqn_realization : public realization {
  list equations;

  void new_equation();

public:
  void menuaction();
  void redraw();
};

class struct_realization : public realization {
  list  submodels;
  list  connections;

  void new_submodel();
  void new_connection();

public:
  void menuaction();
  void redraw();
};
```

Listing 5. Primitive and structured model realizations.

```
class submodel : public component {
  point  position, size;
  model* parent;
  model* sub;
  void*  data;

public:
  void move();
  void scale();
  void instantiate();
  void redraw();
};
```

Listing 6. Definition of the submodel class.

An example

A small example will demonstrate the data structures above: a servo built from a regulator and a motor. On the screen, the engineer will see a block diagram as in Figure 1. Input to the servo is the reference value, also called the setpoint. Output from the servo is the actual position of the actuator. The regulator controls the motor, but the common feedback loop has been left out to simplify the example.

The textual representation in Figure 2 reveals the most important C++ objects needed for the servo. The servo object has two terminals and a realization (terminals and connections will not be described in more detail). The realization is of course structured, and contains two submodels. It also contains three connections: the reference value imported to the regulator, the control signal from regulator to motor (shown in Figure 2), and the exported actuator position.

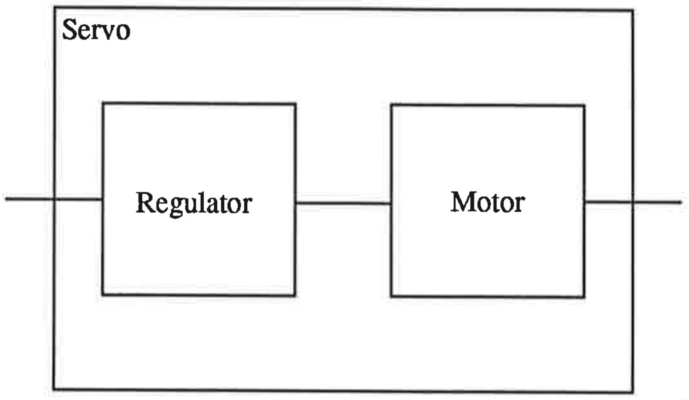


Figure 1. A servo with two submodels.

The submodel objects (for example, **MotorSub**) serve two purposes in this example. Firstly, the graphical appearance of a structured realization is determined mainly by the position and size of the submodels. This information cannot be stored in the model object; a certain kind of motor can be used as a submodel in many different models. Secondly, the submodels establish a relationship between the enclosing model (the servo), and the model objects used as components (e.g., the motor). The two pointers in the submodel object are used, for example, when defining connections. The references between models, realizations and submodels are shown graphically in Figure 3. The role of the submodel when simulating the control system is not discussed here.

The C++ objects used for representing the regulator and the motor are similar to the servo objects. The main difference is that the regulator and the motor have primitive realizations, probably expressed with differential equations.

Exception handling

Handling of exceptions (errors and similar uncommon events) is a problem in all software systems. Ordinary programming techniques, using status flags and if-statements, lead either to bad program structure and cluttered code, or to programs that take proper behaviour for granted. A well designed exception handling mechanism (as in Ada), is an invaluable asset in practical software development. Exceptions increase the readability of the program and indicates the programmer's assumptions about expected and unexpected events [Ghezzi and Jazayeri, 1982, page 22].

The model development tool is quite complex, and many inconsistencies must be checked step-by-step, at different times. Exception handling is useful for restoring the internal data structures to a previous well-defined state. Storing as little redundant information as possible makes this task easier, but may increase complexity in other areas.

The absence of exception handling is a serious flaw of C++. Ada style exception handling, which is also available in C [Lee, 1983], is very effective, but a more flexible scheme may be called for in C++. Some people say that exception handling is needed for developing good class libraries.

Finally, it should be noted that friend functions have been used sparingly (for example, a connection needs free access to terminals and submodels), and proved to be extremely useful. By bending the rules a little, a natural data structure has been maintained; ever-expanding modules because of too strict encapsulation is often a problem with Modula-2 and Ada.

Model: **Servo**
  Terminals: **[Ref, Pos]**
  Realizations: **[ServoRealiz]**

Struct-realization: **ServoRealiz**
  Submodels: **[RegSub, MotorSub]**
  Connections: **[RegSub.u — MotorSub.u, ...]**

Submodel: **RegSub**
  Position: $(-0.6, 0)$
  Size: $(0.5, 0.5)$
  Parent: →**Servo**
  Sub: →**Regulator**

Submodel: **MotorSub**
  Position: $(0.6, 0)$
  Size: $(0.5, 0.5)$
  Parent: →**Servo**
  Sub: →**Motor**

Model: **Regulator**
  Terminals: **[Ref, u]**
  Realizations: **[RegRealiz]**

Eqn-realization: **RegRealiz**
  Equations: [· · ·]

Model: **Motor**
  Terminals: **[u, Pos]**
  Realizations: **[MotorRealiz]**

Eqn-realization: **MotorRealiz**
  Equations: [· · ·]

Figure 2. Textual representation of the servo; terminals, connections and equations are not shown. Square brackets denote a list, an arrow (→) a pointer reference.

5. Using PHIGS

PHIGS (Programmer's Hierarchical Interactive Graphics Standard) is a new 3D graphics standard, aimed at interactive CAE/CAD applications [Brown, 1985]. PHIGS should be regarded as an extension and a complement to the Graphical Kernel Standard [Hopgood et al., 1983], but not as a replacement.

The basic unit in PHIGS is the structure (cf. segment in GKS). A structure contains elements for drawing, graphical attributes, and transformations. It is possible to build hierarchies of structures (i. e., one structure may call another), and to edit the contents of a structure; this is not possible in GKS. Application data may also be stored in a structure, possibly a useful feature.

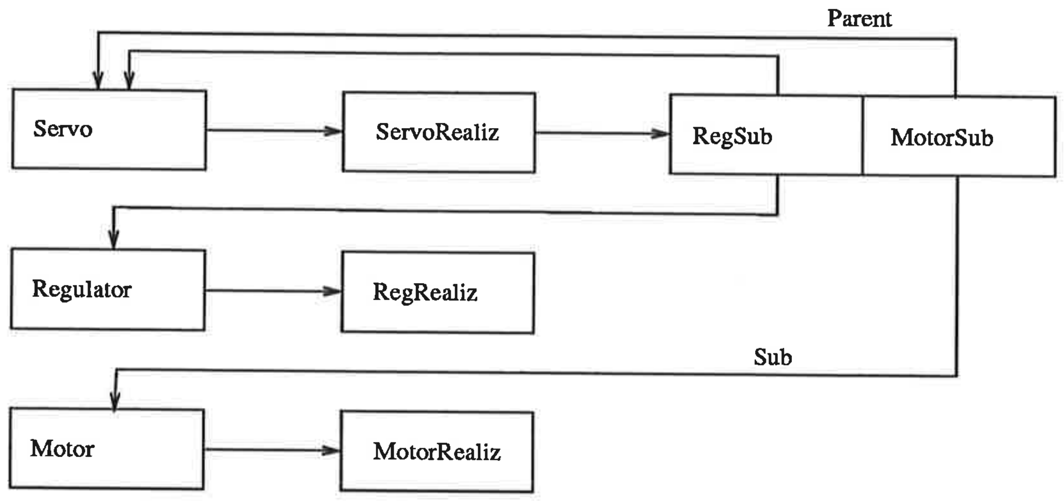


Figure 3. References between models, realizations and submodels of the servo. Terminals, connections and equations are not shown.

In order to take maximum advantage of the hierarchical structures in PHIGS, one structure is associated with every object in the C++ program. This one-to-one correspondence is very convenient; changes are normally localized to a single PHIGS structure, and complete regeneration of the graphics can be avoided. As a typical example, consider changing a pump model: the structure associated with the pump must be changed, but models using the pump as a submodel only refer to a structure identifier, and need no changes. The fine granularity of the graphics hierarchy causes an extra overhead at redraw, which is quite tolerable in this application, though. It can be noted that the model development tool is not a typical PHIGS application, in the sense that it uses the hierarchical features of PHIGS, but not the 3D capabilities.

The correspondence between the object hierarchy and the PHIGS structure hierarchy is shown in Figure 4. The object structure on the left is the same as in Figure 3, but the regulator objects are not shown. A PHIGS structure is associated with each object, as indicated by dashed arrows. The PHIGS structures on the right form a parallel hierarchy, logically connected with "execute structure" primitives. The graphical representation of a model is determined by the realization and its associated structure. The PHIGS structures are in reality more complex, for example, to control picking (see below).

The problem of associating a C++ object with a structure, was solved by some fancy programming. A C++ object can easily refer to a structure by storing the structure identifier, but a problem arises when control must go from a structure to the associated C++ object (for example, when the object's menu action should be invoked). The solution is to use the object's `this` pointer as pick identifier, after conversion to an integer. When the PHIGS system returns a pick identifier, the identifier is converted back to a "pointer to component." The exact nature of the object is not known, but all components implement method `menuaction` (Listing 2).

PHIGS can display graphics on multiple "workstations," which in a workstation environment corresponds to multiple windows. By using so called filters, different graphical representations can be displayed with a single structure hierarchy. Regrettably, multiple workstations are not yet supported by some PHIGS implementations. Event mode input and rubberband lines may also be missing in current implementations. Window management is not available in the PHIGS standard, and may therefore cause considerable practical problems.

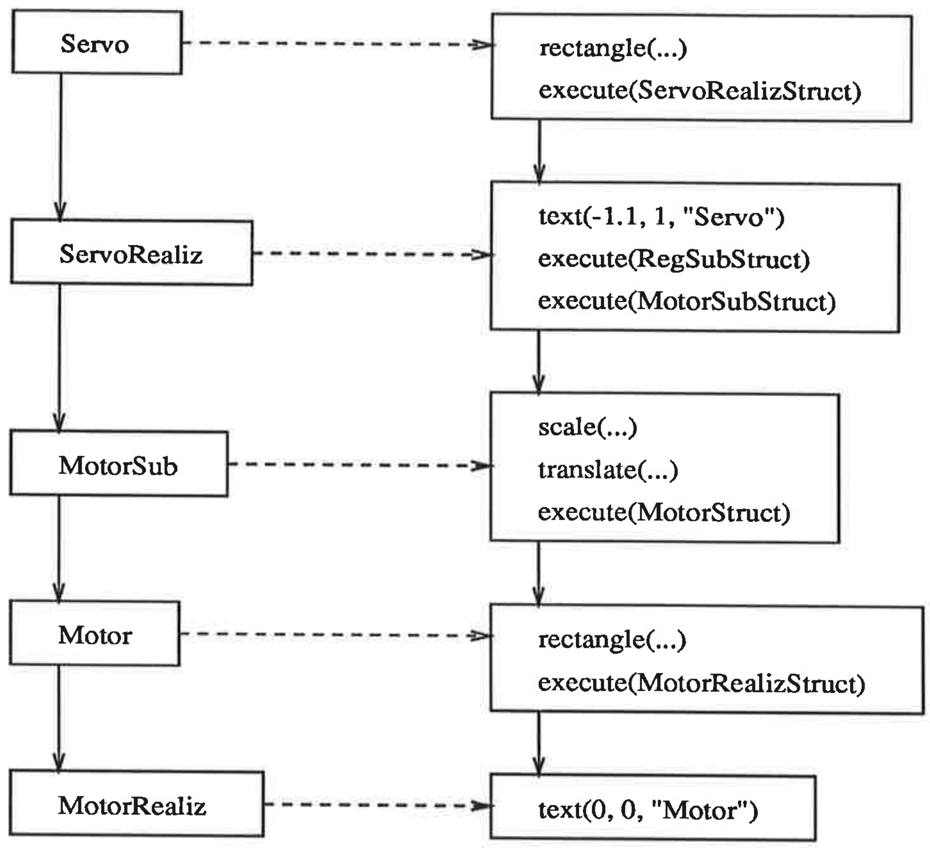


Figure 4. Parallel hierarchies of C++ objects (left) and PHIGS structures (right).

6. Conclusions

In our experience, a dynamic environment like KEE is the best choice for research and rapid prototyping. An engineering tool requires a less expensive and more efficient implementation tool that is available on many computers; in this case, C++ is superior. We have not made a detailed evaluation of KEE versus C++, but the current work shows that programs and data structures using the object-oriented parts of KEE can be implemented in C++ with reasonable effort.

The major difficulty is that C++ does not support dynamic creation of classes. For this reason, models of control systems cannot be directly expressed as classes in C++, so an object-oriented framework must be implemented. The data abstraction and object-oriented programming aspects of C++ provide good support for this framework, and a good programming environment in general. Multiple inheritance, parameterized types and exception handling are much needed extensions to C++.

PHIGS is a powerful new graphics standard, but current implementations need improvement. Window management remains a problem area.

Acknowledgements

I am grateful for many interesting discussions with Sven Erik Mattsson and Mats Andersson, and for comments on the manuscript by Mats Andersson, Ola Dahl and the reviewers. This work was supported by the Swedish National Board for Technical Development (STU).

References

BROWN, MAXINE D. (1985): *Understanding PHIGS*, Template Graphics, San Diego, CA, USA.

GHEZZI, CARLO and MEHDI JAZAYERI (1982): *Programming Language Concepts*, John Wiley & Sons.

HOPGOOD, F. R. A., D. A. DUCE, J. R. GALLOP and D. C. SUTCLIFFE (1983): *Introduction to the Graphical Kernel Standard (GKS)*, Academic Press.

LEE, P. A. (1983): "Exception Handling in C Programs," *Software — Practice and Experience*, **13**, 389–405, May 1983.

MATTSSON, SVEN ERIK (1988a): "On Model Structuring Concepts," *Proc. 4th IFAC Symposium on Computer-Aided Design in Control Systems*, Beijing, P. R. China.

MATTSSON, SVEN ERIK (1988b): "On Modelling and Differential/Algebraic Systems," *Simulation*, Accepted for publication.

NILSSON, BERNT (1987): "Experiences of Describing a Distillation Column in some Modelling Languages TFRT-7362, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden,".

STROUSTRUP, BJARNE (1987a): "The Evolution of C++: 1985 to 1987," *Proc. USENIX C++ Workshop*, Santa Fe, NM, USA.

STROUSTRUP, BJARNE (1987b): "Possible Directions for C++," *Proc. USENIX C++ Workshop*, Santa Fe, NM, USA.

För presentation på TOOLS´89, Paris, November 13-15, 1989

Experiences of Object-Oriented Development in C++ and InterViews

Dag M. Brück
Department of Automatic Control
Lund Institute of Technology
P. O. Box 118
S–221 00 Lund, SWEDEN
E-mail: dag@control.lth.se

Abstract. This paper describes our experiences with InterViews, an object-oriented package for implementing user interfaces written in C++. A comparison is made with PHIGS, a more conventional graphics standard. A strong interaction between base classes and derived classes is observed, notably base classes depending on the behaviour of the derived classes. The application is an interactive block diagram editor. It is used as a stand-alone graphical tool which generates equations for Simnon, a simulator for non-linear systems.

Keywords: Object-oriented programming, User interfaces, C++, InterViews, Computer Aided Control Engineering

1. Background

Developing real control systems is always a difficult task. Mathematical models and simulations are often used in the design and analysis of control systems. The use of computers for this purpose is called Computer Aided Control Engineering (CACE). The development of new CACE tools requires research on the basic concepts of modelling control systems, and the computer representation of control system models [Mattsson, 1988][Andersson, 1989]. Equally important is the choice of tools for developing these tools.

For research and prototyping, a combination of KEE and Common Lisp has proved effective. KEE provides a dynamic and interactive object-oriented development environment, including simple graphical output [IntelliCorp, 1986]. A practical engineering tool for designing control systems must be more economical than an expert system like KEE, so a leaner implementation is needed. C++ is a very good implementation language in this case because of its efficiency and support for object-oriented programming [Brück, 1987].

One of the remaining problem areas is the implementation of the user interface. The developer must choose among a few window managers and several graphics packages. In the on-going evaluation of different alternatives, this paper describes our experiences with InterViews [Linton and Calder, 1987], an object-oriented library for implementing user interfaces, written in C++ [Stroustrup, 1986] and running on the X Window System [Poutain, 1989].

The evaluation of InterViews was conducted by developing a block diagram editor for Simnon, a simulator for non-linear systems [Elmqvist et al., 1986]. Simnon is an interactive, command driven simulation package with its roots in the 1970's; Simnon is still very much state-of-the-art for continuous simulation, but has no graphical input. The block diagram editor is not integrated with Simnon, and therefore reasonably sized for evaluation purposes. This paper also contains a comparison with an earlier evaluation of PHIGS (Programmer's Hierarchical Interactive Graphics Standard) in a similar application. A primitive block diagram editor was developed, but without any relation to Simnon [Brück, 1988]. Previous work has also explored continuous panning, scrolling and zooming of block diagrams on a high-performance workstation. The concept of information zooming was introduced, meaning that the information contents of a block changes depending on its size on the screen [Elmqvist and Mattsson, 1989].

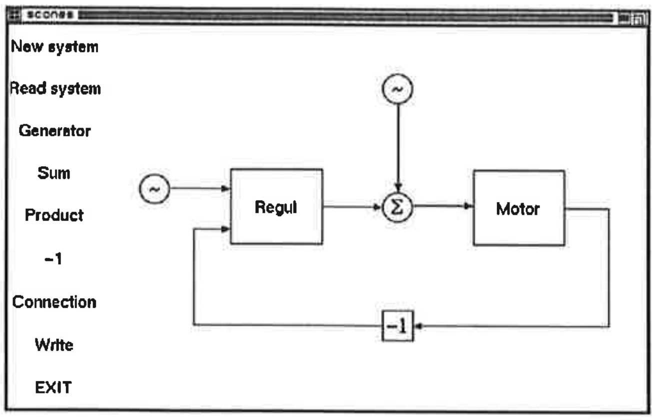


Figure 1. Screen dump of a simple block diagram.

2. The application

A key concept in Simnon is the *system*, which corresponds to a mathematical model of the reality being studied. A system is described in a special modelling language. There are continuous systems based on differential equations and discrete systems based on difference equations. A third type is the connecting system, which is used to form compound systems from enclosed continuous or discrete systems. Every Simnon system is stored as a separate text file.

The connecting system is often visualized (with pencil and paper) by drawing a block diagram. Unfortunately, the drawing must still be transformed into statements of the modelling language. The block diagram editor can produce simple forms of Simnon's CONnEcting System, hence the name *Scones.*

Figure 1 shows a simple block diagram in Scones. There is a fix command menu on the left side, and a drawing area for the block diagram on the right. Systems are represented by large annotated boxes. Special symbols represent the sum (Σ), product (Π, not shown in Figure 1) or negation (-1) of signals. General expressions are represented by generator symbols ($\sim$).

When creating a block diagram, the user can either create a new system in which case Scones will make a template file, or read an existing file in which case Scones will extract properties necessary for drawing the block diagram. Scones knows the name of the system, and maintains for each system a list of terminals (inputs and outputs) which can be connected to terminals of other systems. The connections define the interaction between the systems enclosed by the connecting system. A sequence of connected special symbols are transformed into an arithmetic expression in the connecting system. It should be noted that Scones completely ignores the equations that define the behaviour of a continuous or discrete system. Scones also defines a global time variable t in every connecting system.

The block diagram in Figure 1 represents a servo constructed from a motor and a regulator. A generator provides a step in the regulator's reference value y_{ref}. The control signal from the regulator y is influenced by a load disturbance from another generator. The measured value from the motor u is negated. The template system for the regulator (without equations) is shown in Listing 1. The connecting system produced by Scones is shown in Listing 2. The template code for the motor is very similar to the code for the regulator in Listing 1, and therefore not shown.

```
CONTINUOUS SYSTEM Regul
"Filename pid.t
"Created Fri Feb 10 14:14:51 1989
INPUT y_ref y
OUTPUT u
END
```

Listing 1. Simnon code for the regulator system. Comment lines begin with a double quote (").

```
CONNECTING SYSTEM Consys
"Filename consys.t
"Created Fri Feb 10 14:30:33 1989
TIME t
"System: Regul
y_ref[Regul] = if t > 0 then 1 else 0
y[Regul] = -y[Motor]
"System: Motor
u[Motor] = u[Regul] + sin(t)
"Generator: if t > 0 then 1 else 0
"Generator: sin(t)
END
```

Listing 2. Simnon code for the connecting system.

3. InterViews and PHIGS

InterViews is an object-oriented user interface package [Linton and Calder, 1987]. It provides the basic building blocks for implementing a wide variety of user interfaces. Basic objects derived from the base class *Interactor* can display a graphical image and accept input events. Composite objects derived from class *Scene* can display a complex image by combining other objects (including scenes).

Scenes defined in InterViews can arrange interactors in many ways: side-by-side horizontally (an HBox) or vertically (a VBox), one stacked above the other (a Deck), or framing an interactor (a Frame). Every interactor has a predefined natural size, but may stretch or shrink within specified limits. This means that a scene can adapt to available space by stretching or shrinking its components. Glue objects can be inserted to improve the layout. Other "high-level" user interface objects are scrollers and panners that change the view of a scene, different types of buttons, pop-up menus, and a string editor.

Comparing InterViews with an established graphics standard such as PHIGS [Brown, 1985] is like comparing apples and oranges; the comparison is interesting though, as either InterViews or PHIGS may be the best alternative in a particular application. Superficially, the similarities are striking: both InterViews and PHIGS provide

- Hierarchical structure of graphics. Complex images are constructed by combining simpler objects.
- Reuse of a graphical object in different contexts, and multiple views of a single object.
- Event mode input.

The main difference is in the degree of "object-orientedness." InterViews is fully object-oriented, whereas PHIGS can be classified as object-based [Wegner, 1987]. Graphical objects in PHIGS (called structures) are manipulated by a fixed set of operations, contain only graphical information, and their storage is managed by the PHIGS runtime system. With InterViews, classes derived from class Interactor add behaviour to graphical objects, and can directly represent the real-world object; no separate graphical object hierarchy is needed.

Interactor objects in InterViews are more "live" than structures in PHIGS. When a graphical object changes, it sends a *Change* message to its parent (enclosing scene). InterViews will then send *Redraw* messages to all affected interactors, including the one that was changed; the interactors draw images that reflect their internal state. Redraw messages are also sent on demand from the window manager, for example, when hidden interactors become visible. With PHIGS, the application program must edit the contents of separate structures. The PHIGS system will generate the image by traversing its internal data structures, either on command from the application program, or "when necessary." It is probably easier to use specialized graphics processors or to distribute processing to intelligent graphics terminals in PHIGS, than it is in InterViews.

Similarly, input events are sent directly to the target interactor in InterViews. In PHIGS, the application program will get the identifier of the target structure and of all ancestor structures of the target. The application program is responsible for identifying related objects in its own world. InterViews also contains a set of PHIGS-like graphical objects, derived from class *Graphic*. Apparently, class Graphic does not handle input events, so interactors were used in this project.

Another important difference is the positioning of objects. PHIGS objects are positioned at (x, y); multiple local coordinate systems may be used. In InterViews, objects are typically positioned relative another object, without bothering about the exact coordinates; the object may in fact move around or be reshaped as available space increases or decreases. The InterViews approach is normally much more convenient, and interacts better with the window manager. The

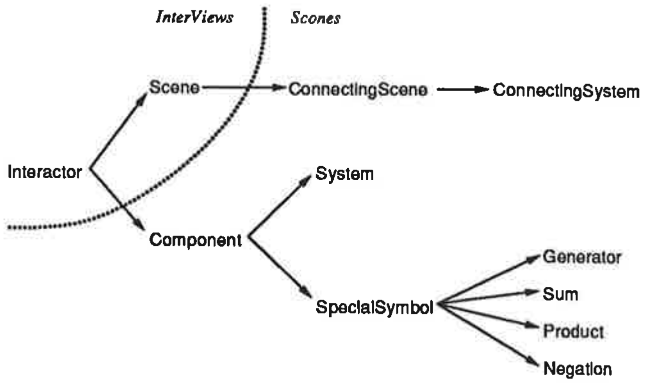


Figure 2. The class hierarchy in Scones.

strengths of PHIGS are its powerful 3D capabilities, and its handling of different projections. Good PHIGS implementations are also significantly more efficient in drawing complex images. Filters are used in PHIGS to control what objects should be visible, pickable, or highlighted. Filters are hardly needed in InterViews, as the graphical image is generated by user written routines that easily adapt to the properties of the corresponding objects. In PHIGS, filters are quite useful.

In short, PHIGS can be regarded as a powerful standard for drawing graphics, and InterViews as a powerful tool for building user interfaces.

4. System design

Scones was designed with simplicity and ease of implementation in mind. It has few features and the user interface is simple. Interaction is mouse based, except for input of text strings. The use of Scones is strongly influenced by the way you draw block diagrams manually, the modelling concepts in Simnon, and the user interfaces of other drawing programs. Internal operation is event driven, using the default event dispatcher of InterViews. Scones was implemented entirely with InterViews and there are no direct calls to the underlying X Window System.

An important objective was closeness to InterViews. Most classes used for representing the block diagram were designed as step-wise augmentations of predefined InterViews classes. Mixing attributes related to the real-world objects being modelled and the graphical attributes is appropriate in this application; in other applications it may be desirable to separate the graphical aspects, for example, to take advantage of distributed graphics processors. The availability of multiple inheritance would probably lead to a design with looser coupling between graphical and modelling aspects. It would then be possible to build a class hierarchy based on the modelling aspects, inheriting graphical aspects as needed from InterViews.

The major class hierarchy in Scones is shown in Figure 2. *Interactor* is the base class of all graphical objects.

Class *Component* represents the common behaviour of all objects in a block diagram. Typical attributes are terminals (the endpoints of a connection), operations on all terminals of a component, and handling of events. Component is an abstract base class (no objects can be directly instantiated) and most operations are realized in derived classes, for example, to generate the equations of the connecting system by following connections. Class *System* represents a continuous or discrete system in Simnon. One specialization is the ability to fire up the editor on the corresponding Simnon text file. The main purpose of all other components is to tie together connections. These attributes are represented by class *SpecialSymbol*, but geometrical shape and arithmetic meaning are realized by derived classes. Class *Generator* has more features than other special symbols (e.g., it can be edited), and should probably have been derived directly from class Component.

A scene in InterViews is essentially an arranger of other objects; this definition also applies to the connecting system of Simnon. The properties most closely related to the operation of InterViews are collected in class *ConnectingScene*. Additional properties related to connections and the generation of equations were collected in class *ConnectingSystem*. The division into two levels of derivation was motivated by the problems in realizing all the needed behaviour of an InterViews scene. Connections are not regarded as objects like systems or summation symbols (and are therefore not interactors), but rather as an attribute of the connecting system. This distinction is probably wrong; many operations (e.g., deletion) would be easier to implement if connections were represented by interactors. The user interface could also be improved if connections responded to mouse clicks.

5. Experiences

The first question that arises when you start using a new software package is "What can I do?" The second is "How should I do it?" Graphics with InterViews can be realized in three complementary ways:

1. InterViews provides a rich set of ready-to-use building blocks, for example, text messages, buttons, and a string editor. These standard interactors are easy to use, easy to integrate (e.g., to create an input form), and behave as expected. □

2. Simple user defined graphical objects are derived from class Interactor. A few low-level methods must be implemented, such as, Redraw. Certain attributes of the interactor must be initialized by the user, for example, the shape object and interest in input events.

The methods and attributes of the low-level objects are not difficult to understand separately, but their use should be better documented to the benefit of new users. When no output at all is produced, it may not be obvious that the real cause was forgetting to initialize the shape member variable. Misuse of attributes, failure to implement a method, or performing initializations in the wrong method, may initially pass unnoticed; in some other context, tried and "debugged" classes may fail for some unexpected reason. □

3. Composite graphical objects that contain other interactors are called scenes. InterViews provides many useful types of scenes, but apparently not a scene that simply puts an interactor at position (x, y) which was needed in Scones.

Implementing a scene is considerably more complex than just drawing some graphics. Firstly, the scene must manage a collection of inserted interactors. A number of operations may require interaction with the enclosed objects, for example, shape calculations. Secondly, the derived scene interacts intimately with its base class and the low-level routines of InterViews. The user written scene must provide a number of services for insertion, deletion, changes, reconfigurations, reshaping, etc. Furthermore, the scene must have a fairly complete set of operations to be operable at all; few shortcuts are possible. On the other hand, once done it is quite easy to comprehend, and not too difficult to redo for a different application. □

Object-oriented programming is apparently more complicated than normally presented, i.e., as simply inheriting behaviour from the base class, or as the base class providing a template for the interface of derived classes. This application shows a strong coupling between base class and derived class; in particular, the function of the base class relies on a properly implemented derived class. This is exactly why the keyword `protected` was introduced into C++; to distinguish class members that must be accessed by derived classes, but not by code outside these classes.

The problems with strong coupling are common in any application where code is reused, and obviously not typical for object-oriented programming. The need for high-quality design and documentation of generally used base classes is pronounced. Object-oriented programming does make it easier to reuse existing code but the designer of a useful base class must anticipate future needs, for example, by declaring methods `virtual` in C++. One may say that object-oriented programming will give you less trouble with the past, and more trouble with the future.

InterViews is a well-designed package, and most problems are due to lack of documentation (about the average UNIX standard). A major improvement would be a description of the internal operations of InterViews, e.g., in the form of a data flow graph. This would give more insight in the interaction between objects, and the intended use of certain methods — what happens when a window is resized? Currently, a major source of documentation is the InterViews code. There are a number of overview papers related to InterViews [Linton and Calder, 1987][Vlissides and Linton, 1988][Linton et al., 1989], and a lively mailing list on Internet.

Little effort was needed to learn how to use InterViews and implement an acceptable user interface, compared to our previous experiences with PHIGS in a similar application. The new user interface is also much improved. Object-oriented programming is well suited to implementing user interfaces, and this application is close to the basic concepts in InterViews. The possibility to express objects directly in C++ and InterViews is a significant advantage, and probably one reason why InterViews is easier to use than PHIGS. Numerous revisions of the program has shown that it is easy to extend the user interface and to add new graphical objects. A considerable amount of time was spent on restructuring existing classes. Two features of InterViews have not yet been evaluated: perspectives for changing the view of a graphical object, and persistent graphics for saving graphical objects on a file.

A reasonable block diagram editor has been implemented in three months, including time to learn InterViews. Scones contains 987 lines of header files (mostly class declarations) and 2347 lines of other code. Users find the program somewhat slow, but it is unclear whether this is because of deficiencies in InterViews or in the X server.

6. Conclusions

InterViews is a powerful object-oriented package for implementing user interfaces. It provides a set of ready-to-use building blocks (e.g., text messages, buttons, a string editor), and simple graphical objects are relatively straight-forward to implement. Non-standard composite graphical objects are considerably more difficult, mainly because of missing documentation.

PHIGS is more efficient and has powerful 3D primitives, but PHIGS is not tailored at implementing user interfaces. Comparing two similar applications, InterViews is easier to use and yields a better user interface.

A surprising experience was the strong interaction between base classes defined in InterViews and derived classes defined in the application. The derived classes not only inherit behaviour, they must also provide services to the base classes and the InterViews system. This coupling stresses the need for good documentation, in particular documentation aimed at the class developer.

Acknowledgements

I am grateful for comments on the manuscript by Sven Erik Mattsson, Mats Andersson, Andrew Koenig and Mark Linton. This work was supported by the Swedish National Board for Technical Development (STU).

References

Andersson, Mats (1989): "An Object-Oriented Modelling Environment," *Proc. 1989 European Simulation Multiconference,* June 7–9 1989, Rome, Italy.

Brown, Maxine D. (1985): *Understanding PHIGS,* Template Graphics, San Diego, CA, USA.

Brück, Dag M. (1987): "Implementation Languages for CACE Software," TFRT-3195, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Brück, Dag M. (1988): "Modelling of Control Systems with C++ and PHIGS," *Proc. USENIX C++ Conference,* October 17–20 1988, Denver, CO, USA.

Elmqvist, Hilding, Karl Johan Åström and Tomas Schönthal (1986): *Simnon User's Guide for MS-DOS Computers,* Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist, Hilding and Sven Erik Mattsson (1989): "Simulator for Dynamical Systems Using Graphics and Equations for Modeling," *IEEE Control Systems Magazine,* **9**, 1, January 1989.

IntelliCorp (1986): *KEE Software Development System User's Manual,* IntelliCorp, Mountain View, CA, USA.

Linton, Mark A. and Paul R. Calder (1987): "The Design and Implementation of InterViews," *Proc. USENIX C++ Workshop,* November 9–10 1987, Santa Fe, NM, USA.

Linton, Mark A., John M. Vlissides and Paul R. Calder (1989): "Composing User Interfaces with InterViews," *IEEE Computer,* **22**, 2, February 1989.

Mattsson, Sven Erik (1988): "On Model Structuring Concepts," *Proc. 4th IFAC Symposium on Computer-Aided Design in Control Systems,* August 23–25 1988, Beijing, P. R. China.

Poutain, Dick (1989): "The X Window System," *BYTE,* January 1989, 353–360.

Stroustrup, Bjarne (1986): *The C++ Programming Language,* Addison-Wesley Publishing Company, Reading, MA, USA.

Vlissides, John M. and Mark A. Linton (1988): "Applying Object-Oriented Design to Structured Graphics," *Proc. USENIX C++ Conference,* October 17–20 1988, Denver, CO, USA.

Wegner, Peter (1987): "Dimensions of Object-Based Language Design," *Proc. OOPSLA'87,* October 4–8 1987, Orlando, FL, USA.

Strukturerad Modellering av Kemiska Processer

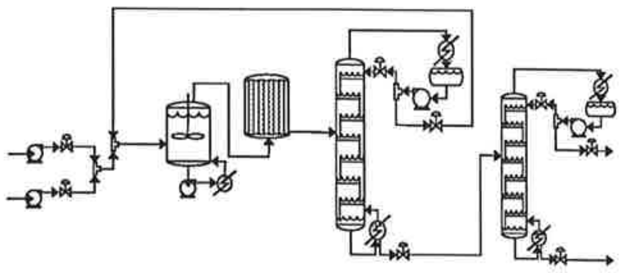
—

Ett Objektorienterat Synsätt

Bernt Nilsson

1. Karakteristiska egenskaper.
2. Objektorienterad processmodellering.
3. Potentialen i objektorienterad modellering.
4. Visioner om datorstödd modellering.

Exempel på kemisk process



Kemiska processer

Karakteristiska egenskaper:

- Många komponenter i komplex struktur.
- Många komponenter är lika.
- Några komponenter är unika.

Processens livscykel:

- Projektering.
- Utbildning, träning och uppstart.
- Drift och underhåll.

Processmodeller

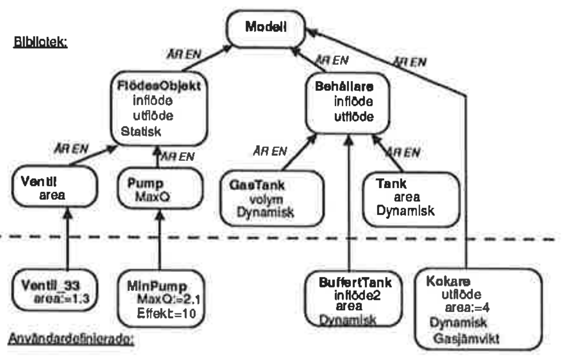
Återkommande processkomponenter

- Sparas i bibliotek.
- Säker återanvändning.
- Anpassningsbara.
  - Parameterisering.
  - Specialisering.

Unika processkomponenter

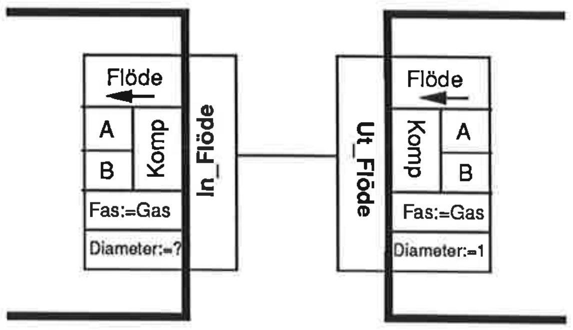
- Specialisering av biblioteksobjekt.
- Återanvändning av "halvfabrikat".
- Modellutveckling med återanvändning av modelldelar.

Utveckling och återanvändning

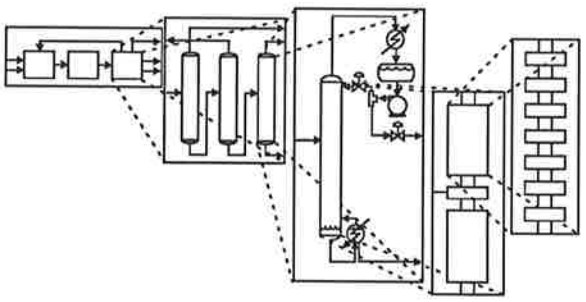


Sammansatta objekt

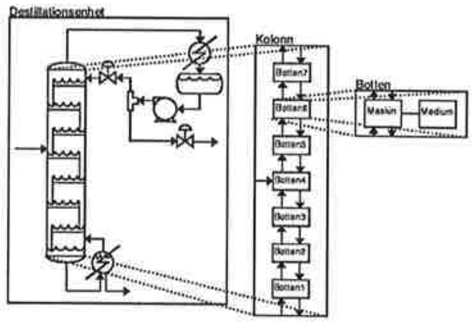
- Automatisk konsistenskontroll.
- Användarspecialisering.
- Parameteröverföring.



Hierarkisk modularisering av processtrukturen



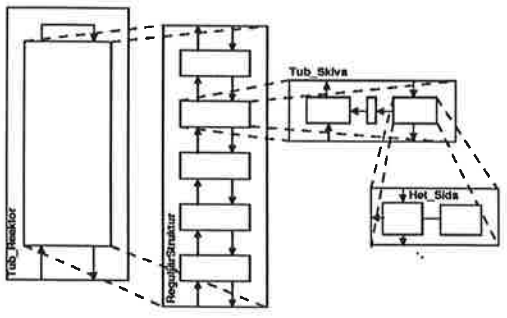
Destillationskolonn



Hierarkisk modularisering ger oberoende parameterisering:

- Konfiguration av enheten.
- Design av kolonnen.
- Maskinmodell (balansekvationer).
- Mediamodell (fasjämvikt).

Tubreaktor

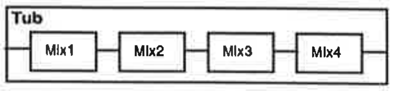


Hierarkisk modularisering ger:

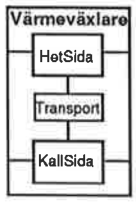
- Möjligheter till återanvändning.
- Underlättad modellutveckling.
- Enkel modellapproximation.

Processpecifik modularisering

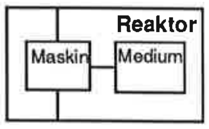
- Modellapproximation



- Transportfenomen

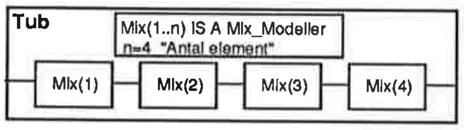


- Media/Maskin

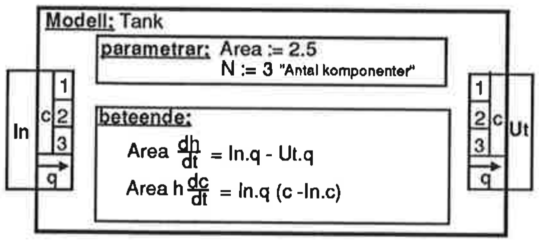


Processpecifik parameterisering

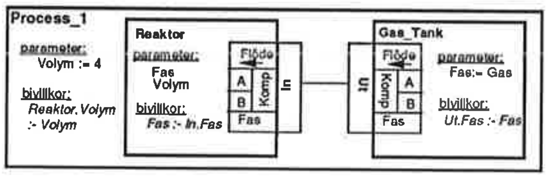
- Strukturparameterisering I



- Strukturparameterisering II


Modell: Tank
parametrar: Area := 2.5
N := 3 "Antal komponenter"
beteende:
Area $\frac{dh}{dt}$ = In.q - Ut.q
Area h $\frac{dc}{dt}$ = In.q (c -In.c)
In
c
1
2
3
q
Ut

- Parameteröverföring och hierarkisk parameterisering.



Objektorienterad Processmodellering

Modellrepresentationen underlättar:

- Modellutveckling.
- Modellåteranvändning
- Modellförfining, modifiering och underhåll.

Gränssnitt:

- Bibliotek
- Modelleditorer.

Användarkrav

Biblioteksutvecklaren:

- Modelleringsmetodik
  - Struktureringskoncept
  - Ärvningsmekanismer
  - Parameteriseringsmetoder

Processmodellutvecklaren:

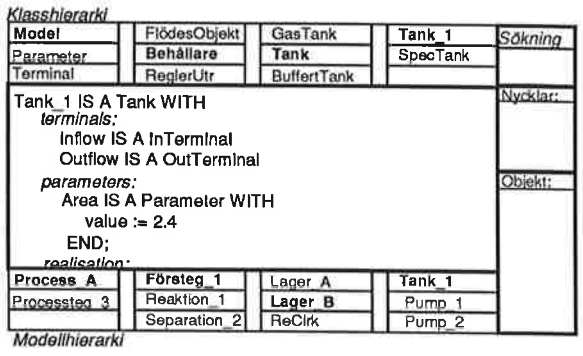
- Lättanvänt bibliotek
  - Modulariserade modeller
  - Avancerad parameterisering
- Speciella modelleditorer

Slutanvändaren:

- Naturligt gränssnitt
- Enkel kommmunikation

Modellbibliotek

- Mycket stort antal objekt.
- Olika utvecklare och användare.



Klasshierarki

Model	FlödesObjekt	GasTank	**Tank_1**	*Sökning*
Parameter	**Behållare**	**Tank**	SpecTank	
Terminal	ReglerUtr	BuffertTank		

```
Tank_1 IS A Tank WITH
   terminals:
      Inflow IS A InTerminal
      Outflow IS A OutTerminal
   parameters:
      Area IS A Parameter WITH
         value := 2.4
      END;
   realisation:
```

Nycklar:

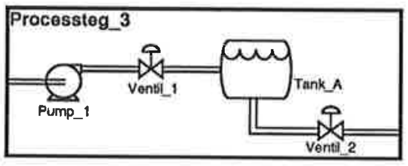
Objekt:

Process_A	**Företeg_1**	Lager_A	**Tank_1**
Processteg_3	Reaktion_1	**Lager_B**	Pump_1
	Separation_2	ReCirk	Pump_2

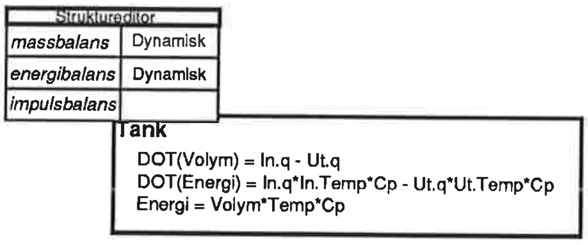
Modellhierarki

Modelleditorer

Sammansatta modeller:



Primitiva modeller:



Strukureditor

massbalans	Dynamisk
energibalans	Dynamisk
impulsbalans	

Tank

```
DOT(Volym) = In.q - Ut.q
DOT(Energi) = In.q*In.Temp*Cp - Ut.q*Ut.Temp*Cp
Energi = Volym*Temp*Cp
```

För presentation på "Annual AIChE Meeting 1989",
San Francisco, Nov 5-10, 1989

Structured Modelling of Chemical Processes with Control Systems

Bernt Nilsson
Department of Automatic Control, Lund Institute of Technology
Box 118, S-221 00 Lund, Sweden
Phone +46 46 108796, Usenet: Bernt@Control.LTH.Se

Abstract. In this paper we discuss an object-oriented approach to modelling of chemical processes with control systems. The basic elements in object-oriented modelling methodology are modularization, model encapsulation, hierarchical submodel decomposition, model parameterization and inheritance. Models have an internal structure of model components, like terminals, parameters and behaviour descriptions. The model behavior can be described with equations or as a connected structure of submodels. Models and model components are represented as objects in single inheritance object class hierarchies. Chemical processes and control systems can be described with the same basic concepts. The object-oriented model representation is implemented in a prototype called System Engineering Environment, SEE. The SEE architecture allows different tools to operate on the models. Tasks that are facilitated in object-oriented modelling are model reuse and model development.

Keywords: Computer simulation; computer-aided design; modeling; process control; process models.

1. Introduction

In this paper we are going to discuss an object-oriented approach to modelling of chemical processes with control systems. Benefits of this approach are facilitated model development and model reuse. It is also possible to adapt and refine models to capture new conditions and demands.

Model structuring concepts are the key to create a modelling environment with these benefits. The models have a given internal structure of model components. An object-oriented approach to modelling represents both models and model components as objects. Modularization, decomposition, parameterization and inheritance are the basic elements in this object-oriented modelling methodology.

A new environment for system engineering (SEE) has been designed with these model structuring concepts. The basic design is composed of a model database, model/user interface and tools that operate on models. One tool, that is implemented in a prototype, is a simulator for differential and algebraic equations. This architecture allows an object-oriented approach to model development and an equation-oriented approach to the problem solving. SEE is presented in Mattsson and Andersson (1989), Andersson (1989a) and in Nilsson et al (1989).

This paper is organized as follows: An example of a process model is discussed in Section 2. Object-oriented modelling and model structuring concepts are introduced in Section 3. Modelling of controlled chemical process is discussed in Section 4 and in Section 5 are some conclusions.

2. The Tank Reactor Example

The main ideas are illustrated on a minor chemical process part, namely an exothermic continuous stirred tank reactor. The reactor is assumed to be homogeneous in concentration and temperature. A chemical reaction is assumed to occur, $A \rightarrow B$, and it produces heat. The reactor vessel

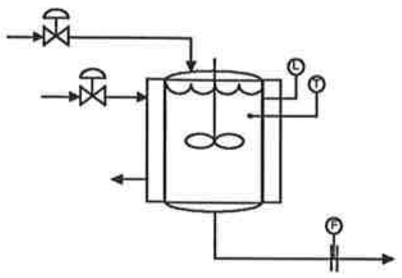


Figure 1. The continuous stirred tank reactor.

can now be modelled with a dynamic mass balance, dynamic component mass balances and an energy balance. The feed to the reactor is controlled by a valve. The outflow of the reactor is set by the surrounding system. Heat is removed by a cooling jacket. The cooling jacket can be assumed to be homogeneous and modelled by a dynamic energy balance. The cooling medium flow is also controlled by a valve.

The mathematical model of the reactor system then becomes a set of nonlinear differential equations:

$$\rho \frac{dV}{dt} = \rho q_{in} - \rho q_{out}$$
$$\frac{d(Vc)}{dt} = q_{in} c_{in} - q_{out} c + Vr$$
$$\rho C_p \frac{d(VT)}{dt} = \rho C_p q_{in} T_{in} - \rho C_p q_{out} T - Q$$
$$\rho_j C_{p_j} \frac{d(V_j T_j)}{dt} = \rho_j C_{p_j} q_j T_{j_{in}} - \rho_j C_{p_j} q_j T_j + Q$$

$$r_1 = -r_2 = -k_0 e^{-\frac{E_a}{RT}} c_1 \quad ; \quad Q = \kappa A(T - T_j)$$

The concentration, c, and reaction velocity, r, are column vectors with the length of two, to describe the components A and B.

The out flow of the tank is described by a static momentum balance, which means that the flow is a function of the height in the tank and the pressure drop over the tank:

$$\rho g h + p_{tank} = \frac{(q_{out}/a)^2}{2} + p_{outlet}$$

The pressure drop over the valves can be modelled by a static momentum balance too. The pressure drop is a function of the flow and the valve position:

$$\Delta p = \frac{Ku}{2A^2} q_{in} |q_{in}| \quad ; \quad 1 \geq u \geq 0$$

The tank reactor model described above is equation-oriented. Model representations, like this, do not have any structure. The model is hard to reuse in new applications. It is not easy to change the model and it is hard to read and understand the model for a unexperienced user.

3. Object-Oriented Modelling

In object-oriented modelling models are represented as objects. Object-oriented modelling is based on the methodology from object-oriented programming. A good introduction to object-oriented programming is given in Stefik and Bobrow (1984).

An object-oriented model representation has been design in the SEE-prototype (Andersson, 1989a). A textual language for the model representation is called Omola, Object-oriented Modelling Language (Andersson, 1989b).

In this section we are first discussing some *model structuring concepts* and then the *inheritance concept.*

Internal Model Structure

A model object has an internal structure of model component objects. The internal structure of a model is composed of three major component types:

1. *Terminal* is a model component which can be used to describe interaction with a connected model.
2. *Parameter* is a model component that allows the user to interact with the model, in order to adapt its behaviour to new applications.
3. *Behaviour description* or realization is a description of the model behaviour. The behaviour can be primitive, expressing the behaviour symbolically with equations, or it can be composite and described by a structure of connected submodels. Models can have multiple realizations.

Model structuring concepts are disussed in more detail by Mattsson (1988). Model structures are also discussed in Åström and Kreutzer (1986) and in Åström and Mattsson (1987).

An object-oriented model representation of the reactor vessel can be seen in Figure 2. It is composed of terminals, parameters and primitive behaviour description. There are five terminals and seven parameters. The behaviour is described by dynamic mass, component and energy balances, which are the same as the first three differential equations in Section 2, one static momentum balance.

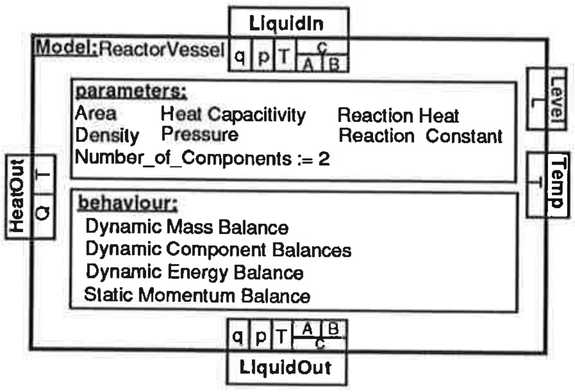


Figure 2. The internal structure of the reactor vessel model.

Submodel Interaction

In Figure 3 the structure of the reactor system is shown and it is described as a composite model object. The different parts are modularized into submodel objects. The connections between submodels represents submodel interactions. The interaction between two submodels is given by the terminal descriptions. If a connection is drawn between two terminals then the system make a consistency check. The two terminals on each side of the connection must have the same internal structure.

Terminals with internal structures, that describe a pipe connection, are the **LiquidIn** and **LiquidOut** in Figure 2. Connections can have natural interpretations, like the one between the tank reactor model, TankReactor, and the Valve1 model object. In a mathematical model this connection represents a set of relations between variables. This means that flow (q), pressure (p), temperature (T) and concentration (c) in **TankReactor** and in **Valve1** are set equal or summed to zero. This kind of submodel interaction is well documented by Mattsson (1989). Terminals are defined as objects. This means that an process pipe terminal class can be a super-class of every process pipe terminal object in the process model. Terminal descriptions are therefore easy and natural to reuse.

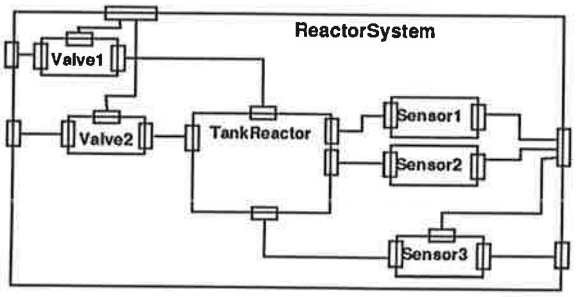


Figure 3. A block diagram showing the composite model of the tank reactor system.

Hierarchical Submodel Decomposition

The tank reactor model can be decomposed into three submodels, namely one reactor vessel model (ReactorVessel), one cooling jacket model (Jacket) and one heat transfer model (HT-model). The reactor vessel model is a primitive model and is seen in Figure 2. This means that the tank reactor model is a composite model, with three submodels, and we get a hierarchy of models. This is a hierarchical submodel description and it is shown in Figure 4.

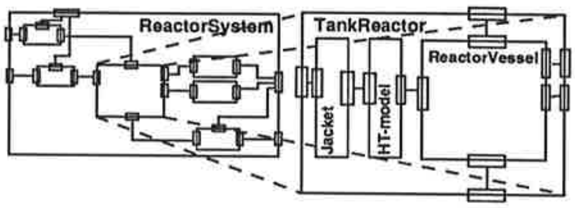


Figure 4. The hierarchical submodel decomposition in the tank reactor example.

This decomposition makes it possible to reuse submodels that is not directly interpretated as physical components. A heat transfer model object is an example of this. It is possible to change the heat transfer model without changing surrounding submodels or the super-model structure.

Inheritance

Models are represented as objects, which are subclasses of predefined super-classes. A subclass inherits properties from its super-class. The model representation has single inheritance, which means that a subclass only has one superclass. The properties that are inherited are the object attributes, which are definitions of components. Model object inherits model component definitions. A system defined super-class Model is the root of the model class hierarchy tree and a specialization means that attributes defining model components are added to the subclass.

An example of how to use the inheritance concept is shown in Figure 5. The class Valve is a subclass of the system defined super-class Model. It is specialized by getting two attributes that define two model components. These model components are two terminals describing the inflow and the outflow of the valve object. Valve is a super-class to ControlValve, which have two additional attributes describing the control signal terminal and a parameter. The two valves used in the reactor system are specializations of ControlValve. They contain specializations of the parameter attribute Area.

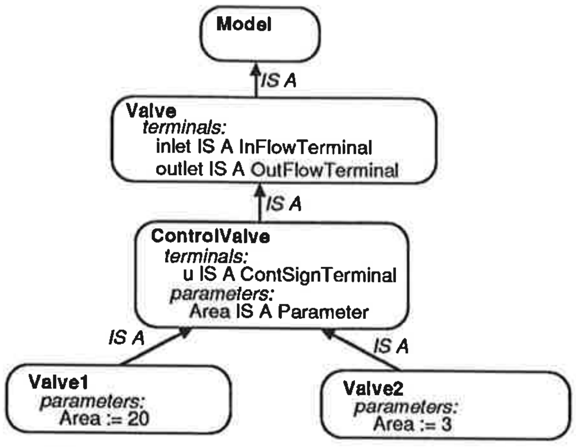


Figure 5. A part of model class hierarchy tree describing the relation between some valve models.

Parameterization

Parameterization of models and model components are important in our attempt to reuse objects. Design variables are defined as parameters, which can be changed by the user. Area and density are examples in the reactor vessel model in Figure 2. *Structure parameterization* of the reactor vessel model means that the dimension of vectors, like concentration, are being set by a parameter. The reactor vessel can be reused in a new application with another number of chemical components by changing this parameter.

An important method of parameterization is to decompose the reactor vessel model into one vessel machine model and one chemical medium model, a *medium and machine decomposition.* The machine model contains the main behaviour description (balance equations) and machine parameters (area). The medium model contains the medium behaviour (reaction velocity) and medium parameters (reaction heat and density).

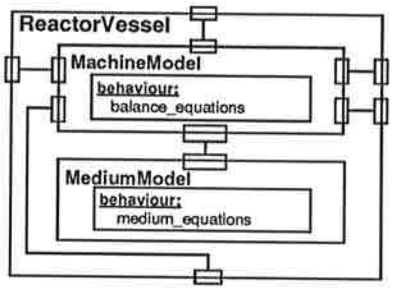


Figure 6. A medium and machine decomposed reactor vessel model.

The reactor vessel model can be decomposed into two submodels, which are connected to each other, which is seen in Figure 6. This can be seen as a parameterization of the reactor vessel. Another reactor vessel model can be created through inheritance of the attributes from the old one. The medium model can be change by overwriting the medium model definition.

A Modelling Methodology

A modelling methodology can use decomposition, parameterization and inheritance to create process models that are generic and easy to reuse.

Decomposition of process models into smallare submodels is important for abstraction of the modelling problem. Different decomposition methods are process structure decomposition into process objects, see Figure 3, transport phenomenon decomposition, like in the tank reactor model in Figure 4, or the medium and machine decomposition seen in Figure 6. The resulting submodels are often basic descriptions of fundamental behaviours.

Inheritance can be used as a model type concept and support reuse of similar models and model components. Also by overwriting inherited attributes can models be modified in order to create new models.

Parameterization of models should be made to suit the user and facilitate reused. Changing a parameter of a reused model is done by overwriting the inherited parameter value with a new value. The definition of a submodel can be changed in a similar way by overwriting the old submodel definition. One important application of this is the overwriting of media model definitions.

Decomposition and parameterization methods and the use of inheritance and discussed in Nilsson (1989).

4. A Controlled Chemical Process

We have seen how one can use an object-oriented approach to the modelling of the tank reactor process. We are now focusing on the control system description.

The Controlled Tank Reactor

The control system for the tank reactor process can be described in a similar way. The reactor has one structured terminal describing the control signal of the two valves. It has also one structured terminal describing the three sensors: level, temperature and outflow. A model of the control system is connected to the reactor system through the control signal terminal and the sensor measurement terminal. This is seen in Figure 7. The control system is a composite model with

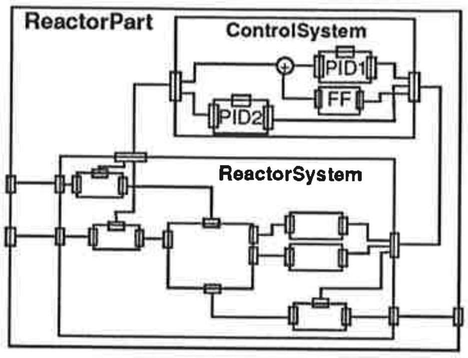


Figure 7. The tank reactor with control system.

an internal structure of submodels that represents the different controllers.

The first control system design is based on two PID-controllers, which is seen in Figure 6. One controller (PID1) uses the inflow valve to control the reactor level. The outflow measurement is used for feed forward control of the level. The other PID (PID2) controls the reactor temperature through the cooling medium valve.

Modification of the Control System

A modification of the control system is easy to do. A second control system design can be a MIMO-controller based on a LQG-design on state-space form. The state feedback and observer submodels are subclasses of generic classes with a parameterization that facilitates reuse. In this case these are specialized to capture a system with three inputs and two outputs. In an environment with tools for symbolic and numeric manipulations we can first symbolically linerize the model into a linear model and then use the numerical tool to calculate a LQG-controller, which automatically create a controller like the one in Figure 8.

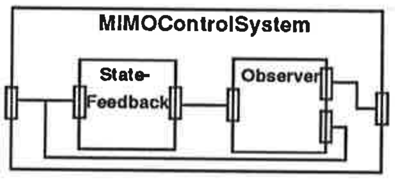


Figure 8. A state-space based MIMO control system.

This MIMO controller can now be used in the reactor part model. In a reactor part model the old definition of the control system can be overwritten by the definition of the new one. The new control system model must have terminal with the same internal structure as the old one.

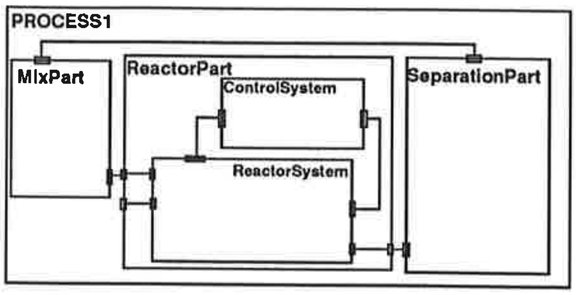


Figure 9. A chemical plant model that reuse the reactor process part model.

Plant Models

The resulting composite model, ReactorPart, can be reused in its turn in a chemical process plant model. One example is the Process1 shown in Figure 9. It is now possible to study the the control system on a complete plant model. If we have models for other parts of the process then it is easy to connect them together to create a plant model. A study of the new control system design can now be done based on realistic disturbances from the surrounding equipment.

Multiple Presentations

Large processes with control systems can be seen in a number of different ways. One way is the *process oriented view* where the controllers are distributed all over the process in order to fit the process structure description. Another way is the *control system oriented* block diagram view where the feedback loop are the most important. A third view is *computer oriented* where the hardware and software are in focus. All these views are different presentations of the same model representations. It should be possible to have different presentations of the same object.

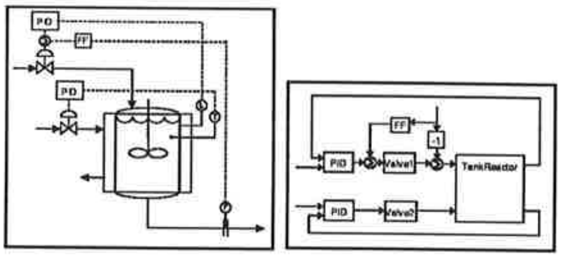


Figure 10. Two different presentations of a controlled chemical process. Left: a process oriented view. Right: a control system oriented view.

User interfaces for simulation also needs multiple presentations. A control engineer and a process operator need different interfaces. It is important to have interfaces that are natural and convenient for the user. The user interface presentations does not have to have the same structure as the model representation.

5. Conclusions

Model reuse, development, refinement and maintenance are facilitated through the concepts of modularization, decomposition, parameterization and inheritance.

Model Reuse

The strong modularization concept with encapsulated submodels with terminals supports easy and safe reuse of models. Decomposition of models into submodels makes it possible to reuse the structure and change the submodels in the structure. Advanced parameterization of models can increase the reusability of models. Inheritance means that the model object description can be distributed in a tree of super-classes and can therefore be reused.

Model Development

Model development is facilitated in three ways. One is the possibility to reuse submodels from model libraries. Predefined submodels can be reused in new composite models describing new applications. This is possible due to the strong modularization. One example is to use common process equipments, like pumps, valves etc., to create a complex process.

The possibility to decompose a process model in a multiple level description facilitate development of complex systems. The model developer can chose the amount of abstraction on each level.

Another way to facilitate model development is to use the inheritance and specialize predefined objects to describe new models in new applications. This way to develop models is of major importance and has a great potential. This is shown in the tank reactor example.

Model Refinement and Maintenance

To adapt and to modify model behaviour to real plant data requires methods for model refinement and long term use of process models requires possibilities to change, reuse and refine models. A model class can have multiple realizations and this can be used to refine the behaviour of models. A model can first get a simple behaviour description. It can then easily be refined by getting an additional behaviour description. The behaviour descriptions can be static, dynamic, simple, complex, linear or nonlinear. The user can choose a desired realization depending on the application. Model maintenance also requires readable and easierly changeable models, which are facilitated by decomposition into small objects and by inheritance.

6. Acknowledgements

I would like to thank Karl Johan Åström for initializing this work. I would also like to thank Mats Andersson and Sven Erik Mattsson, who are behind the object-oriented modelling architecture, for many and long useful discussions.

7. References

Andersson, M. (1989a): "An Object-Oriented Modelling Environment," *1989 European Simulation Multiconferance, ESM'89, Rome, Italy.*

Andersson, M. (1989b): "Omola - An Object-Oriented Modelling Language," Report TFRT-7417, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Åström, K.J. and W. Kreutzer (1986): "System Representations," *IEEE Third Symposium on Computer-Aided Control System Design, Arlington, Virginia.*

Åström, K.J. and S.E. Mattsson (1987): "High-Level Problem Solving Languages for Computer Aided Control Engineering," Report TFRT-3187, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Mattsson, S.E. (1988): "On Model Structuring Concepts," *4th IFAC Symposium on Computer-Aided Design in Control Systems, P.R. China.*

Mattsson, S.E. (1989): "Modeling of Interactions between Submodels," *1989 European Simulation Multiconferance, ESM'89, Rome, Italy.*

Mattsson, S.E. and M. Andersson (1989): "An Environment for Model Development and Simulation," Report TFRT-3205, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Nilsson, B. (1989): "Structured Modelling of Chemical Processes – An Object-Oriented Approach," Lic Tech Thesis TFRT-3203, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Nilsson, B, S.E. Mattsson and M. Andersson (1989): "Tools for Model Development and Simulation," *Proceedings of the SAIS '89 Workshop.*

Stefik, M. and D.G. Bobrow (1984): "Object-Oriented Programming: Themes and Variations," *The AI Magazine, Vol. 6, No. 4.*

ITM

Institutet för tillämpad matematik
The Swedish Institute of applied mathematics

Differential – Algebraic Equations in fifteen minutes!

by

Gustaf Söderlind

ITM Tel. 08 / 10 95 47

WHAT ARE DAE's ?

DAE's are coupled differential and "algebraic" equations

Ex

$$\begin{cases} \dot{x} = f(t, x, y) \\ 0 = g(t, x, y) \end{cases}$$

Ex

$$F(t, x, \dot{x}) = 0 \quad \text{with}$$

$$\det F_{\dot{x}} = 0$$

The dimension of the state space is less than the number of equations.

NOTE! The dimension of the state space depends on the number AND structure of the "algebraic" equations!

MODELS AND PROBLEMS

A model is an underdetermined system:

- a set of variables
- a set of equations

Ex Newton's second law

$$\begin{cases} m\dot{v} = F \\ \dot{x} = v \end{cases}$$

Variables: x, v, F

Parameters: m

A problem for the model is specified by selecting data, so that one can solve for the remaining variables.

PROBLEM = MODEL + DATA

DIRECT AND INVERSE PROBLEMS

Ex Dynamics (Direct problem)

$$\begin{cases} m\dot{v} = F \\ \dot{x} = v \\ 0 = F - F_R \end{cases} \quad \begin{matrix} \left.\begin{matrix} \\ \\ \end{matrix}\right\} \text{model} \\ \left.\right\} \text{problem} \end{matrix}$$

F_R given as a function of t, x, v

Ex Robotics (Inverse problem)

$$\begin{cases} m\dot{v} = F \\ \dot{x} = v \\ 0 = x - x_R \end{cases} \quad \begin{matrix} \left.\begin{matrix} \\ \\ \end{matrix}\right\} \text{model} \\ \left.\right\} \text{problem} \end{matrix}$$

x_R is a prescribed trajectory

The dynamics problem is easily solved numerically (index 1). The robotics problem is difficult (index 3).

The Index of a DAE

A DAE may differentiate its input.

Consider

$$\begin{cases} F(t, x, \dot{x}) = 0 & x(0) = x_0 \\ F(t, y, \dot{y}) = r(t) & y(0) = x_0 \end{cases}$$

If $\|x - y\| \leqslant C \cdot \left(\max\|r\| + \max\|r'\| + \ldots + \max\|r^{(m-1)}\| \right)$

then index = m.

- Pure integration $\Rightarrow$ index 0
- Integration + constant perturbation $\Rightarrow$ index 1
- One differentiation $\Rightarrow$ index 2
- Two differentiations $\Rightarrow$ index 3

Robotics example has index 3.

The index can be reduced by differantiating the equations and eliminating variables.

In the "robotics" problem,

$$\begin{cases} m v_{n+1} = m v_n + h F_{n+1} \\ x_{n+1} = x_n + h v_{n+1} \\ 0 = x_{n+1} - x_R(t_{n+1}) \end{cases}$$

we obtain

$$\begin{pmatrix} v_{n+1} \\ x_{n+1} \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & -h^{-1} & \\ & 0 & \\ -mh^{-1} & -mh^{-2} & 0 \end{pmatrix} \begin{pmatrix} v_n \\ x_n \\ F_n \end{pmatrix} + x_R(t_{n+1}) \begin{pmatrix} h^{-1} \\ 1 \\ mh^{-2} \end{pmatrix}$$

Note negative powers of h!

Can convergence as $h \to 0$ be ascertained?

The problem has index 3 and differentiates the data $x(t)$ twice: the (generalized) force which is being computed is $F = m\ddot{x}$.

Why is it Difficult to Solve High-Index Problems?

Apply Backward Euler method to our dynamics problem:

$$\begin{cases} m v_{n+1} = m v_n + h F_{n+1} \\ x_{n+1} = x_n + h v_n \\ 0 = F_{n+1} - F_R(t_{n+1}) \end{cases}$$

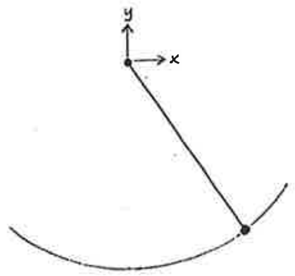
h = timestep, $t_{n+1} = t_n + h$.

Then,

$$\begin{pmatrix} v_{n+1} \\ x_{n+1} \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & & \\ h & 1 & \\ & & 0 \end{pmatrix} \begin{pmatrix} v_n \\ x_n \\ F_n \end{pmatrix} + F_R(t_{n+1}) \begin{pmatrix} h/m \\ h^2/m \\ 1 \end{pmatrix}$$

Thus, in this index 1 problem v, x and F change by $\mathcal{O}(h)$ in a single step. This is the "normal" situation.

The Inevitable Mathematical Pendulum



Unit mass, unit length pendulum

$$\begin{cases} \dot{x} = u \\ \dot{y} = v \\ \dot{u} = -\lambda x \\ \dot{v} = -\lambda y - g \\ 0 = x^2 + y^2 - 1 \end{cases}$$

1 DOF motion
index 3

λ represents constraint forces.

This problem cannot be treated satisfactorily using today's numerical solution techniques.

Sources of DAE's

- **Electrical networks**

  Inductors, capacitors $\Rightarrow$ ODE
  Resistors $\Rightarrow$ DAE

  Normally index ≤ 2

- **Mechanics**

  Systems of rigid bodies
  Constrained motion $\Rightarrow$ DAE

  Normally index $= 3$

- **Fluid dynamics**

  Incompressible viscous flow

  Normally index $= 2$ after spatial discretization

- **Process dynamics**

  Index depends on model structure and problem formulation

Important:

Input/Output - free (or assignment-free) models

I/O modelling $\Rightarrow$ specific problem

I/O - free $\Rightarrow$ Descriptor models, DAE's

Example

Components: $E_{0i}\,\dot{x}_i = f_i(t, x_i, u_i)$

Collection of components:

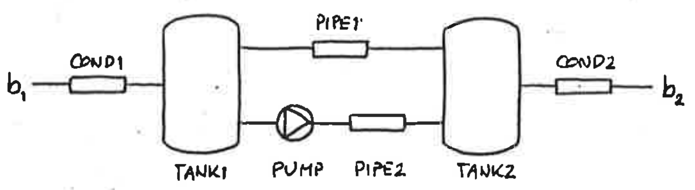
$$E_0\,\dot{x} = f(t, x, u)$$

Component interactions:

$$0 = E_x x + E_u u + b(t)$$

E_x, E_u contain $+1$, -1 as nonzeros.

A simple test problem



Conductance: $q = UA(T_1 - T_2)$

Tank: $mc_p\dot{T} = q + qf_1 + qf_2$

$k\dot{p} = w_1 + w_2$

Pipe: $w_1 = f\cdot(p_1 - p_2)$

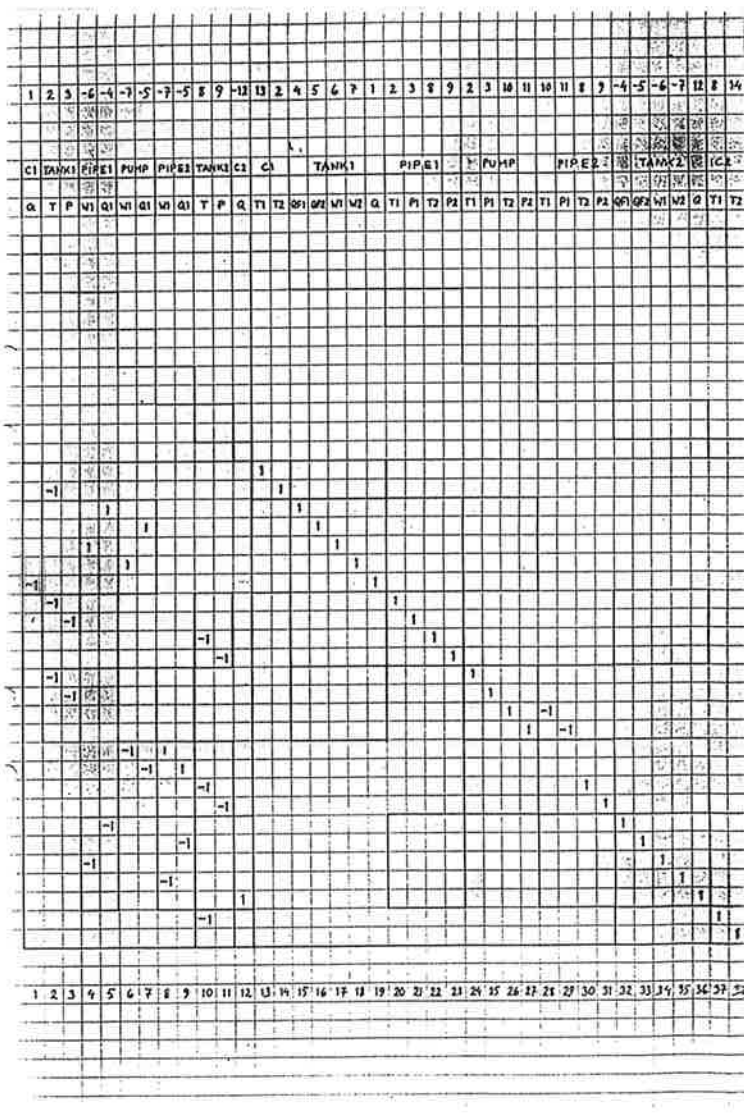
$q_1 = w_1 T_1 + UA(T_1 - T_2)$

Pump: $w_1 = \text{switch}\cdot a\cdot(1 + (p_2 - p_1)(b + c\cdot(p_2 - p_1)))$

$q_1 = w_1 T_1$

Boundary data: $b_1 = \text{COND1}.T_1$

$b_2 = \text{COND2}.T_2$



Framtida behov för modell-
utveckling och simulering i
massa- och pappersindustrin

Sven Gunnar Edlund
STFI

Dagsläge beträffande utnyttjande av simulering

o Konstruktion/projektering
  - etablerat

o Utbildning
  - visst utnyttjande

o Träning
  - under introduktion

o Beslutsstöd
  - visst utnyttjande

Kritiskt för att nå ökad användning av simulering:

BRA PROCESSMODELLER

Behövs även för styrning

Processerna karakteriseras av:

- fysikaliska/kemiska samband komplexa och delvis okända
- både långsam och snabb dynamik
- olinjära, flervariabla
- stora brister i observerbarhet
- processambanden förändras över tiden

Exemplet: Beslutsstöd

- o Används uthålligt
  - kunna lita på modellen
  - efter 300 gångers användning kan svaret förutses:

    $\longrightarrow$ detaljerad modell

    lätt att komplettera modellen
- o Små skillnader mellan de alternativ som simuleras
  - kräver noggrann modell
- o Modellen måste hållas uppdaterad
  - automatisk/adaptiv bestämning av parametrar/samband
  - effektivt stöd för manuellt modellunderhåll

    $\longrightarrow$ behov av process-analysverktyg

Behov av kraftfulla verktyg/metoder för

- modellutveckling
- modellvalidering
- modellunderhåll

MMC-gränssnitt och modeller måste utformas utgående från användarens behov, arbets-situation och förutsättningar

SSPA
SYSTEMS

MARINTEKNISKA SIMULATORER

Claes Källström, SSPA Systems

Exempel på simulatorer vid Sjöbefälskolor:

- Radarsimulator
- Manöversimulator
- Ballastsimulator
- Maskinrumssimulator

POSTADRESS *POSTAL ADDRESS*	BESÖKSADRESS *STREET ADDRESS*	TELEFON *TELEPHONE*	TELEGRAM *TELEGRAMS*	TELEX	TELEFAX	POSTGIRO *POSTAL CHEQUE ACCOUNT*	BANKKONTO *BANK ACCOUNT*	BANKGIRO *BANK GIRO*
Box 24001 S-400 22 Göteborg Sweden	Chalmers Tvärgata 10 Göteborg	Nat 031-63 95 00 Int +46-31 63 95 00	MARITIME	20863 SSPAGBG S	inl +46 31 63 96 24	462 52-3	SE-BANKEN 5027-100 21 90	152-4875

SSPA SYSTEMS

Exempel på marinmilitära simulatorer :

- Radarsimulator
- Maskinrumssimulator
- Stridsledningssimulator
- Ubåtssimulator

Övrigt :

- Segelbåtssimulator

Staffan Nordmark, VTI 1989-09-18

Förberedande inlägg i paneldiskussion: **Framtida behov för modellutveckling och simulering**

VTI (Statens Väg- och Trafikinstitut) bedriver forskning inom vägtrafikområdet och simuleringsteknik och simulatorer har under lång tid varit hjälpmedel i denna forskning. Det har handlat om allt från rena digitalkörningar, där tidsfaktorn inte är viktig, till hybridsimuleringar i realtid där verkliga komponenter används för en del av systemet och resten beskrivs i ett datorprogram. Många av de program som utvecklades på skilda håll i världen under 60- och 70-talen kunde kräva flera manår i utveckling och man kunde i stort sett vara säker på att programkoden innehåller flera felaktigheter pga de komplicerade ekvationerna. Validering kan i de flesta fall säkerställa att felen åtminstone är försumbara. Självklart är det en stor fördel om dessa felfunktioner kan elimineras och program kan konstrueras av andra än programmeringsspecialister.

För mekaniska problem finns en klar tendens att använda stora simuleringspaket (multi-body systems) såsom ADAMS, MEDYNA, DADS osv för att underlätta modellbyggandet och slippa tidsödande härledningar av rörelseekvationer. Med dessa programpaket kan den tid som åtgår för modellkonstruktion och programmering drastiskt skäras ner men till priset av långa exekveringstider och stor datakapacitet.

Det vore önskvärt om motsvarande utveckling även kan ske i realtidssammanhang. Det är ställt utom allt tvivel att användning i en simulator av kördynamiska program är en utmärkt validering i sig. Som förare har man goda möjligheter att relatera till verkligheten och avslöja felaktigheter i programkoden eller modelluppbyggnaden. Realtidsbegränsningen gör emellertid att för närvarande är de generella programpaketen uteslutna även om vissa ansatser åt detta håll har gjorts av Daimler Benz och Evans & Sutherland för simulatortillämpningar. Generaliteten är begränsad och insatsen av datorer avsevärd men detta kommer givetvis att förändras i framtiden.

KLASSISK SIMULERING

* Besvärliga härledningar av ekvationer
* Stort programmeringsarbete
* Stort antal felkällor
* Kort exekveringstid
* Måttliga datorkrav

SIMULERINGSPAKET

ADAMS, DADS, MEDYNA m.fl.

* enkel användning
* lång exekveringstid
* stor datorkapacitet

SIMULATORER

REALTIDSKRAV

medför

* tidsoptimerade program
* begränsad storlek på programmen eller flera parallellkopplade datorer
* generella programpaket kan användas enbart i begränsad utsträckning

REALTIDSKRAV medför historiskt

<u>- 1975</u>

* analogimaskiner ev. hybridmaskiner med assemblerprogram

<u>1975 - 1980</u>

* hybridmaskiner med digitalprogrammen i högnivåspråk (FORTRAN)

<u>1980 - 199?</u>

* en eller flera parallella digitala datorer med program i högnivåspråk

<u>199? -</u>

* generella programpaket för Multi-body systems