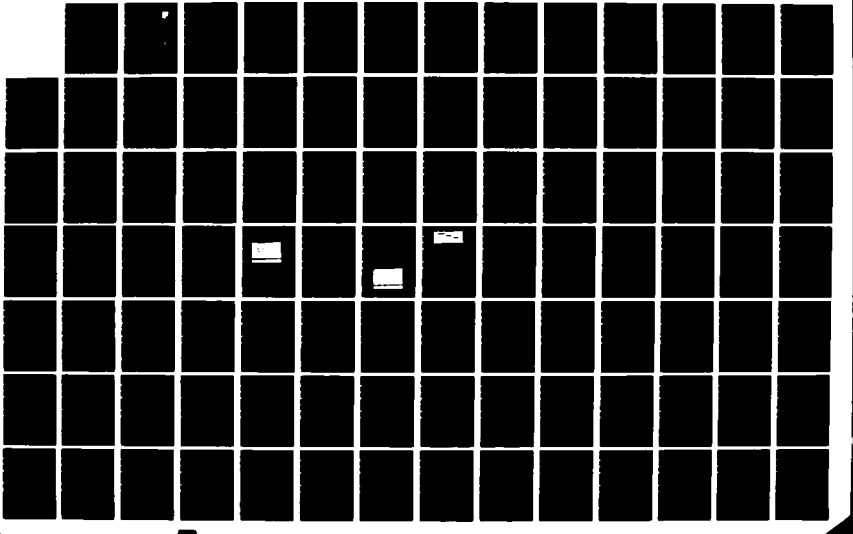
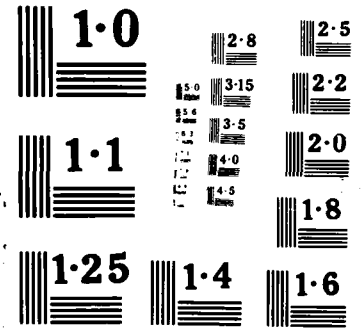
AD-A189 819 ANNUAL KNOWLEDGE-BASED SOFTWARE ASSISTANT CONFERENCE (2ND) HELD ON 18-20 (U) ROME AIR DEVELOPMENT CENTER GRIFFISS AFB NY K M BENNER JAN 88 RADC-TR-87-243 1/4

UNCLASSIFIED F/G 12/5 NL




1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5

(12)

RADC-TR-87-243
Proceedings
January 1988

SECOND ANNUAL KNOWLEDGE-BASED SOFTWARE ASSISTANT CONFERENCE

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-87-243 has been reviewed and is approved for publication.

APPROVED: Kevin M. Benner

KEVIN M. BENNER, 1LT, USAF
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notice on a specific document requires that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A189 819

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS N/A
2a SECURITY CLASSIFICATION AUTHORITY N/A	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-243	5 MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a NAME OF PERFORMING ORGANIZATION Rome Air Development Center 6b OFFICE SYMBOL (If applicable) COES	7a NAME OF MONITORING ORGANIZATION N/A
6c ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	7b ADDRESS (City, State, and ZIP Code) N/A
8a NAME OF FUNDING / SPONSORING ORGANIZATION Rome Air Development Center 8b OFFICE SYMBOL (If applicable) COES	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N/A
8c ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: 62702F; PROJECT NO: 5581; TASK NO: 27; WORK UNIT ACCESSION NO: TK

11 TITLE (Include Security Classification)
SECOND ANNUAL KNOWLEDGE-BASED SOFTWARE ASSISTANT CONFERENCE

12 PERSONAL AUTHOR(S)
Editor: 1Lt Kevin M. Benner

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Proceedings	FROM 8 Aug 87 TO 20 Aug 87	January 1988	308

16 SUPPLEMENTARY NOTATION
N/A

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Artificial Intelligence; Knowledge-Based Programming; Knowledge-Based Systems; Software Assistant; Software Environments; Automatic Programming

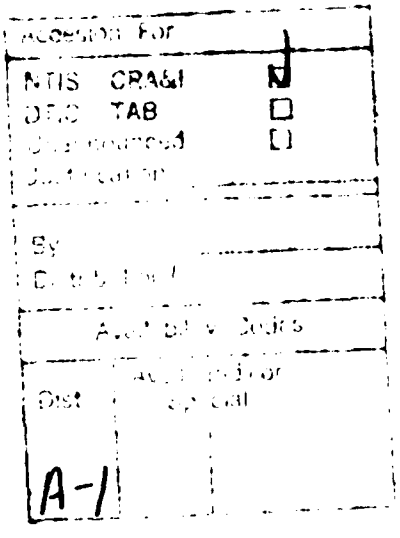
19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This report is the proceedings of the 2nd Annual Knowledge-Based Software Assistant Conference, held 18 - 20 August 1987. This proceeding presents the current state of KBSA research supported by RADC as well as some related work not supported by RADC.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Kevin M. Benner, 1Lt, USAF	22b TELEPHONE (Include Area Code) (315) 330-3564
22c OFFICE SYMBOL RADC (COES)	

DD Form 1473, JUN 86 — Previous editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

TABLE OF CONTENTS



2ND ANNUAL RADC
KNOWLEDGE-BASED SOFTWARE ASSISTANT
CONFERENCE

18-20 August 1987
Utica, NY

MONDAY, 17 August 1987

5:00	PM	to	9:00	PM	**Early Registration & Reception, Sheraton Inn Ballroom Foyer**

TUESDAY, 18 August 1987

8:00	AM	to	9:00	AM	**Registration, Sheraton Inn Ballroom Foyer**
9:00	AM	to	9:05	AM	**Welcome** Lt. Kevin M. Benner, Rome Air Development Center
9:05	AM	to	9:15	AM	**Welcome Address** Col. Richard J. Stachurski, Rome Air Development Center
9:15	AM	to	9:45	AM	**KBSA Overview** Lt. Kevin M. Benner, Rome Air Development Center
9:45	AM	to	10:15	AM	**A Common Framework for Knowledge-Based Programming** Mr. Steven D. Huseth, Honeywell Systems and Research Center
10:15	AM	to	10:45	AM	**Morning Break**
10:45	AM	to	11:15	AM	**An Overview of the Knowledge-Based Requirements Assistant** Mr. David R. Harris, Sanders Associates, Inc.
11:15	AM	to	11:45	AM	**Overview of the Knowledge-Based Specification Assistant** Dr. W. Lewis Johnson, USC-Information Sciences Institute
11:45	AM	to	12:15	PM	**Performance Estimation for a Knowledge-Based Software Assistant** Mr. Allen Goldberg, Kestrel Institute
12:15	PM	to	1:00	PM	**Open Discussion**
1:00	PM	to	2:30	PM	**Lunch**
2:30	PM	to	4:00	PM	**KBSA: Has the Perspective Changed Since the Original Report in 1983?** Dr Robert M. Balzer, USC-Information Sciences Institute Dr. Barry W. Boehm, TRW Dr. Thomas E. Cheatham, Software Options, Inc. Dr. Cordell C. Green, Kestrel Institute Dr. Charles Rich, Massachusetts Institute of Technology

WEDNESDAY, 19 August 1987

9:00	**AM**	**to**	**9:05**	**AM**	**KBSA Technology Transfer, Introduction of Panel** Mr. Samuel A. DiNitto, Jr., Rome Air Development Center Dr. Robert J. Glushko, Software Engineering Institute Mr. Robert W. Lawler, Boeing Computer Services Mr. Kenneth E. Nidiffer, Software Productivity Consortium Dr. William L. Scherlis, Defense Advanced Research Projects Agency
9:05	**AM**	**to**	**9:50**	**AM**	**KBSA Technology Transfer, Panel Members' Statements**
9:50	**AM**	**to**	**10:10**	**AM**	**KBSA Technology Transfer, Panel Discussion**
10:10	**AM**	**to**	**10:25**	**AM**	**Open Discussion**
10:25	**AM**	**to**	**10:55**	**AM**	**Morning Break**
10:55	**AM**	**to**	**11:35**	**AM**	**KBSA Common Framework Implementation** Mr. Aaron R. Larson, Honeywell Systems and Research Center
11:35	**AM**	**to**	**12:15**	**PM**	**Where's the Intelligence in the Intelligent Assistant for Requirements Analysis?** Mr. Andrew J. Czuchry, Jr., Sanders Associates, Inc.
12:15	**PM**	**to**	**1:45**	**PM**	**Lunch**
1:45	**PM**	**to**	**2:25**	**PM**	***Turning Ideas into Specifications*** Dr. W. Lewis Johnson, USC-Information Sciences Institute
2:25	**PM**	**to**	**3:05**	**PM**	Dr. William L. Scherlis, Defense Advanced Research Projects Agency
4:00	**PM**	**to**	**5:30**	**PM**	**KBSA Demonstrations at Rome Air Development Center** **Requirements Assistant** Sanders Associates, Inc. **Framework** Honeywell Systems and Research Center **Project Management Assistant** Kestrel Institute
7:00	**PM**				**Dinner, The Metro**

THURSDAY, 20 August 1987

9:00	AM	to	9:40	AM	**Plans and Meta-Plans in GRAPPLE: An Intelligent Assistant for the Process of Programming** Ms. Karen E. Huff, University of Massachusetts
9:40	AM	to	10:20	AM	**Technical Issues for Performance Estimation** Mr. Allen Goldberg, Kestrel Institute
10:20	AM	to	10:50	AM	**Morning Break**
10:50	AM	to	11:30	AM	**Designing an Effective User Interface for a Knowledge-Based Development Environment** Mr. Timothy G. King, Honeywell Systems and Research Center
11:30	AM	to	12:10	PM	**Knowledge-Based Tools for Knowledge-Based Systems** Dr. Lance A. Miller, IBM Federal Systems Division
12:10	PM	to	1:20	PM	**Lunch**
1:20	PM	to	2:00	PM	**Project Management Assistant: What Have We Learned?** Mr. Louis J. Hoebel, Rome Air Development Center
2:00	PM	to	2:40	PM	**The Knowledge Integration Tool: Knowledge Interface (A Knowledge-Based System Development Environment)** Mr. Philip H. Newcomb, Boeing Advanced Technology Center for Computer Sciences
2:40	PM	to	3:05	PM	**ASDL, An Intelligent Acquisition Support Tool** Dr. Richard M. Adler, The MITRE Corporation
4:00	PM	to	5:30	PM	**KBSA Demonstrations at Rome Air Development Center** **Performance Assistant** Kestrel Institute **GRAPPLE** University of Massachusetts **Specification Assistant** USC-Information Sciences Institute

Mr. Bernard B. Abrams
Grumman Aircraft Systems Division
Mail Station B38-35
Bethpage, NY 11714
(C) (516) 575-8477
(D) ABRAMS@ADA20

Ms. Patricia J. Baskinger
IIT Research Institute
Beeches Technical Campus
Turin Road, Route 26N
Rome, NY 13440
(C) (315) 336-2359
(D) Baskinger@RADC-TOPS20

Dr. Richard M. Adler
The MITRE Corporation
Burlington Road
Mail Stop A155
Bedford, MA 01730
(C) (617) 271-3569

Lt. Kevin M. Benner
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COES
Griffiss AFB, NY 13441-5700
(C) (315) 330-3564 (A) 587-3564
(D) Benner@RADC-Multics

Mr. Bradley P. Allen
Inference Corporation
5300 West Century Boulevard
Los Angeles, CA 90045
(C) (213) 417-7997

Mr. Edward H. Bensley
The MITRE Corporation
Mail Stop A345
Burlington Road
Bedford, MA 01730
(C) (617) 271-2587

Dr. Robert M. Balzer
USC-Information Sciences Institute
4676 Admiralty Way
Suite 1001
Marina del Ray, CA 90292-6695
(C) (213) 822-1511
(D) Balzer@vaxa.isi.edu

Dr. Valdis Berzins
Naval Postgraduate School
Computer Science Department
Monterey, CA 93943
(C) (408) 646-2735 (A) 878-2735

Dr. Steven Barry
TRW
Systems Division
2750 Prosperity Avenue
Fairfax, VA 22031
(C) (703) 876-4170

Mr. Eric R. Beyler
AT&T Bell Laboratories
Room 6C525
Naperville-Wheaton Road
Naperville, IL 60566
(C) (312) 979-4000

Dr. David R. Barstow
Schlumberger-Doll Research
Old Quarry Road
Ridgefield, CT 06877-4108
(C) (203) 431-5501
(D) BARSTOW@SLB-TEST.CSNET

Dr. Barry W. Boehm
TRW
One Space Park
R2/2086
Redondo Beach, CA 90278
(C) (213) 535-2184

Mr. Daniel M. Burstyn
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COES
Griffiss AFB, NY 13441-5700
(C) (315) 330-4833/2601 (A) 587-4833

Mr. Lawrence J. Cristina, Jr.
HQ U.S. Army Strategic Defense Command
ATTN: DASD-H-SBY
PO Box 1500
Huntsville, AL 35807-3801
(C) (205) 895-3846 (A) 742-3846

Dr. Thomas E. Cheatham
Software Options, Inc.
22 Hilliard Street
Cambridge, MA 02138
(C) (617) 497-5054

Mr. Andrew J. Czuchry, Jr.
Sanders Associates, Inc.
MER24-1283, C.S. 2034
144 Daniel Webster Highway-Merrimack
Nashua, NH 03061-2034
(C) (603) 885-9184

Ms. Judith A. Clapp
The MITRE Corporation
Burlington Road
Mail Stop A356
Bedford, MA 01730
(C) (617) 271-3755

Dr. Robert E. Dalton
VITRO Corporation
Mail Stop 1-1328
14000 Georgia Avenue
Silver Spring, MD 20906-2972
(C) (301) 231-1004

Mr. John B. Coon
IBM Corporation
Federal Systems Division
18100 Frederick Pike
Gaithersburg, MD 20850
(C) (301) 240-3642

Mr. Samuel A. DINitto, Jr.
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COE
Griffiss AFB, NY 13441-5700
(C) (315) 330-3011 (A) 587-3011

Mr. Dennis Cooper
General Research Corporation
5383 Hollister Avenue
Santa Barbara, CA 93111
(C) (805) 964-7724

Mr. Donald M. Ellefante
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COTC
Griffiss AFB, NY 13441-5700
(C) (315) 330-3241 (A) 587-3241
(D) Ellefante@RADC-Multics

Mr. Richard E. Copra
Vista Controls Corporation
27825 Fremont Court
Valencia, CA 91355
(C) (805) 257-4430

Ms. Mary Emrich
Oak Ridge National Laboratory
PO Box X
Oak Ridge, TN 37831-6207
(C) (615) 576-8272

Mr. Richard M. Evans
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COES
Griffiss AFB, NY 13441-5700
(C) (315) 330-3564 (A) 587-3564
(D) Evans@RADC-TOPS20.ARPA

Dr. Dennis W. Fife
Institute for Defense Analyses
Computer & Software Engineering Div.
1801 N. Beauregard St.
Alexandria, VA 22311
(C) (703) 845-3512

Mr. Mark Fisher
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COES
Griffiss AFB, NY 13440-5700
(C) (315) 330-3564 (A) 587-3564

Dr. Northrup Fowler III
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COES
Griffiss AFB, NY 13441-5700
(C) (315) 330-7794 (A) 587-7794
(D) Fowler@RADC-Multics

Ms. Maria Francesca
The MITRE Corporation
Mail Stop A180T
Burlington Road
Bedford, MA 01730
(C) (617) 271-3534

Dr. Armen Gabrielian
Thomson-CSF Inc.
630 Hansen Way
Suite 250
Palo Alto, CA 94306
(C) (415) 494-8818

Ms. Nancy M. Giddings
Honeywell Systems and Research Center
3660 Technology Drive
MN65-2100
Minneapolis, MN 55418
(C) (612) 782-7337

Mr. J. Terry Ginn
Sanders Associates, Inc.
MER24-1283, C.S. 2034
144 Daniel Webster Highway-Merrimack
Nashua, NH 03061-2034
(C) (603) 885-9192

Dr. Robert J. Glushko
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213
(C) (412) 268-7795
(D) glushko@sei.cmu.edu

Mr. Wolfhart B. Goethert
IIT Research Institute
Beeches Technical Campus
Turin Road, Route 26N
Rome, NY 13440
(C) (315) 336-2359

Dr. Robert T. Goettge
Advanced Systems Technologies, Inc.
12200 East Briarwood Avenue
Suite 260
Englewood, CO 80112
(C) (303) 790-4242

Mr. Allen Goldberg
Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304-1216
(C) (415) 493-6871

Dr. Cordell C. Green
Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304-1216
(C) (415) 493-6871

Ms. Karen E. Huff
University of Massachusetts, Amherst
29 Westland Avenue
Winchester, MA 01890
(C) (617) 729-0206

Mr. Paul A. Gyergyek
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COEE
Griffiss AFB, NY 13441-5700
(C) (315) 330-2054 (A) 587-2054

Mr. Steven D. Huseth
Honeywell Systems and Research Center
3660 Technology Drive
Software Technology, MN65-2100
Minneapolis, MN 55418
(C) (612) 782-7340
(D) Huseth@HI-Multics

Mr. David R. Harris
Sanders Associates, Inc.
MER24-1283, C.S. 2034
144 Daniel Webster Highway-Merrimack
Nashua, NH 03061-2034
(C) (603) 885-9182/9241

Mr. Kevin D. Jackson
Naval Surface Weapons Center
Code N23
Dahlgren, VA 22448
(C) (703) 663-4472

Mr. William E. Hefley
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213
(C) (412) 268-7793
(D) weh@sei.cmu.edu

Dr. W. Lewis Johnson
USC-Information Sciences Institute
4676 Admiralty Way
Suite 1001
Marina del Rey, CA 90292-6695
(C) (213) 822-1511
(D) johnson@vaxa.isi.edu

Mr. William M. Hodges
Boeing Computer Services
Mail Stop 82-54
PO Box 3999
Seattle, WA 98124-2499
(C) (206) 773-5485

Dr. Richard C. Jullig
Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304-1216
(C) (415) 493-6871

Mr. Louis J. Hoebel
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COES
Griffiss AFB, NY 13441-5700
(C) (315) 330-3564 (A) 587-3564
(D) HOEBEL@RADC-TOPS20.ARPA

Mr. Timothy G. King
Honeywell Systems and Research Center
3660 Technology Drive
Software Technology, MN65-2100
Minneapolis, MN 55418
(C) (612) 782-7670

Dr. Gary B. Lamont
U.S. Air Force Institute of Technology
Dept. of Electrical Engineering and
Computer Science
Mail Stop AFIT/ENG
Wright-Patterson AFB, OH 45433
(C) (513) 255-2057 (A) 785-2057

Mr. Robert Little
British Embassy
3100 Massachusetts Avenue, N.W.
Washington, DC 20008
(C) (202) 898-4618
(D) RLITTLE@A.ISI.EDU

Mr. Aaron R. Larson
Honeywell Systems and Research Center
3660 Technology Drive
Software Technology, MN65-2100
Minneapolis, MN 55418
(C) (612) 782-7308

Mr. Raymond Liu
Naval Ocean Systems Center
Code 423
San Diego, CA 92152-5000
(C) (619) 225-6682 (A) 933-6682
(D) LIU@NOSTECR.ARPA

Prof. Jeffrey A. Lasky
Rochester Institute of Technology
Graduate Department of Computer Science
One Lomb Memorial Drive
PO Box 9887
Rochester, NY 14623-0887
(C) (716) 475-2284

Dr. Luqi
Naval Postgraduate School
Computer Science Department
Monterey, CA 93943
(C) (408) 646-2735 (A) 878-2735

Mr. Robert W. Lawler
Boeing Computer Services
Boeing Advanced Technology Center for
Computer Science
PO Box 24346, MS 7L-43
Seattle, WA 98124-0346
(C) (206) 865-3325

Ms. Ann Marmor-Squires
TRW
Systems Division
2750 Prosperity Avenue
Fairfax, VA 22031
(C) (703) 876-4100

Mr. Martin B. Lewis
Grumman Aerospace Corporation
Software System Engineering Directorate
MS B38-35
Bethpage, NY 11714
(C) (516) 575-7862

Mr. Richard J. Mayer
Texas A&M University
Knowledge Based Systems Laboratory
College Station, TX 77843-3317
(C) (409) 845-9363

Ms. Valerie Lipman
General Dynamics
Electronics Division
PO Box 85310
Mail Stop 7202-K
San Diego, CA 92138-5310
(C) (619) 573-7246

Mr. W. L. McCoy
Naval Surface Weapons Center
Code K54
Dahlgren Laboratory
Dahlgren, VA 22448
(C) (703) 663-8367 (A) 249-8367
(D) WMCCOY@NSWC

Mr. Robert E. Meliot
Unisys Defense Systems
Suite 1100
8201 Greensboro Drive
McLean, VA 22102
(C) (703) 847-3248

Ms. Penny Muncaster-Jewell
Texas Instruments, Inc.
Computer Science Center
PO Box 655474
Mail Stop 238
Dallas, TX 75265
(C) (214) 995-3750
(D) muncaster%ti-csl@csnet-r

Dr. Lance A. Miller
IBM Corporation
Federal Systems Division
18100 Frederick Pike
Mail Stop 501
Gaithersburg, MD 20879
(C) (301) 240-6743

Mr. Phillip H. Newcomb
Boeing Computer Services
Advanced Technology Center for
Computer Sciences
PO Box 24346, Mail Stop 7L-64
Seattle, WA 98124
(C) (206) 865-3431

Mr. James R. Milligan
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COEE
Griffiss AFB, NY 13441-5700
(C) (315) 330-4063 (A) 587-4063

Mr. Kenneth E. Nidiffer
Software Productivity Consortium
1880 Campus Commons Drive, North
Reston, VA 22091
(C) (703) 391-1818

Ms. Marilyn S. Mills
Eastman Kodak
Management Services Division
Kodak Park Site B-56
Rochester, NY 14650
(C) (716) 477-3449

Ms. Mary Anne Overman
National Security Agency
NSA Computer System Software
ATTN: T303
9800 Savage Road
Fort Meade, MD 20755-6000
(C) (301) 688-7805
(D) Overman@MIT

Mr. Michael P. Minette
Hughes Aircraft Company
Software Engineering Division
PO Box 3310
Mail Stop 618/M215
Fullerton, CA 92634-3310
(C) (714) 732-2822

Dr. Charles Rich
MIT Artificial Intelligence Laboratory
545 Technology Square
Room 839
Cambridge, MA 02139-1986
(C) (617) 253-7877

Mr. John G. Morris
Texas A&M University
Knowledge Based Systems Laboratory
College Station, TX 77843
(C) (409) 845-9363

Mr. Thomas R. Robbins
IIT Research Institute
Data & Analysis Center for Software
RADC/COED
Griffiss AFB, NY 13441-5700
(C) (315) 336-0937 (A) 587-3395
(D) Robbins@RADC-Multics

Mr. Donald F. Roberts
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COES
Griffiss AFB, NY 13441-5700
(C) (315) 330-2973 (A) 587-2973

Mr. James L. Sidoran
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COEE
Griffiss AFB, NY 13441-5700
(C) (315) 330-2762 (A) 587-2762
(D) SIDORAN@RADC-SOFTVAV

Dr. A. Joseph Rockmore
Advanced Decision Systems
201 San Antonio Circle
Suite 286
Mountain View, CA 94040-1289
(C) (415) 941-3912

Col. Richard J. Stachurski
U.S. Air Force Systems Command
Rome Air Development Center
RADC/CV
Griffiss AFB, NY 13441-5700
(C) (315) 330-7701 (A) 587-7701

Mr. Robert N. Ruberti
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COES
Griffiss AFB, NY 13441-5700
(C) (315) 330-3528 (A) 587-3528

Ms. Nancy L. Sunderhaft
IIT Research Institute
Data & Analysis Center for Software
RADC/COED
Griffiss AFB, NY 13441-5700
(C) (315) 336-0937 (A) 587-3395
(D) Sunderhaft@RADC-Multics

Mr. Jay Rumkel
Sanders Associates, Inc.
MER24-1283, C.S. 2034
144 Daniel Webster Highway-Merrimack
Nashua, NH 03061-2034
(C) (603) 885-9235

Mr. Linwood Sutton
Naval Ocean Systems Center
Code 423
San Diego, CA 92152-5000
(C) (619) 225-6682 (A) 933-6682
(D) SUTTON@ADA20.ISI.EDU

Mr. William E. Rzepka
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COEE
Griffiss AFB, NY 13441-5700
(C) (315) 330-2762 (A) 587-2762

Mr. Phillip C. Topping
Lockheed Missile and Space Company
2710 Sand Hill Road
Menlo Park, CA 94025
(C) (415) 354-5255

Dr. William L. Scherlis
Defense Adv. Research Projects Agency
DARPA/ISTO
1400 Wilson Boulevard
Arlington, VA 22209
(C) (202) 694-5800

Dr. Walter Tsai
University of Minnesota
Department of Computer Science
Minneapolis, MN 55455
(C) (612) 625-6371

Mr. Thomas W. Tuning
Martin Marietta Data Systems
6100 Bandera Road
Suite 900
San Antonio, TX 78238-1696
(C) (512) 522-1100

Mr. Gregory M. Whittaker
The MITRE Corporation
RADC/TOFM
Building 106
Griffiss AFB, NY 13441-5700
(C) (315) 336-4966 (A) 587-4277
(D) GW@MITRE-bedford.arpa

Mr. Raymond P. Urtz, Jr.
U.S. Air Force Systems Command
Rome Air Development Center
Command and Control Directorate
RADC/CO
Griffiss AFB, NY 13441-5700
(C) (315) 330-2165 (A) 587-2165

Mr. Richard C. Wyatt
The MITRE Corporation
RADC/TOFM
Building 106
Griffiss AFB, NY 13441-5700
(C) (315) 336-4966

Dr. Ralph Wachter
Office of Naval Research
Information Science Program
800 North Quincy Street
Arlington, VA 22217
(C) (202) 696-4304

Dr. Donald P. Yu
SYSCON Corporation
5792 Widewaters Parkway
Dewitt, NY 13214
(C) (315) 445-2066

Mr. Kurt C. Wallnau
Unisys Corporation
Paoli Research Center
Route 252 and Central Avenue
PO Box 517
Paoli, PA 19301
(C) (215) 648-7458

Mr. Douglas A. White
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COES
Griffiss AFB, NY 13441-5700
(C) (315) 330-3564 (A) 587-3564
(D) White@RADC-Multics

Dr. Stephanie M. White
Grumman Aircraft Systems Division
Mail Stop B38-35
Bethpage, NY 11714
(C) (516) 575-2297

Addendum

2nd Annual KBSA Conference
18-20 August 1987

Ms. Susan C. Lilly
IBM
18100 Frederick Pike
Gaithersburg, MD 20879
(C) (301) 240-7178

Paula S.D. Mayer
Texas A&M University
Knowledge Based Systems Laboratory
College Station, TX 77840
(C) (409) 845-5030

Mr. David A. Vennergrund
TRW
Federal Systems Group
2750 Prosperity Avenue
Fairfax, VA 22032
(C) (703) 876-4304

THE KNOWLEDGE-BASED SOFTWARE ASSISTANT

Overview

Lt Kevin M. Benner, USAF and Douglas A. White
Command and Control Technology Division
Rome Air Development Center
Griffiss AFB, NY 13441-5700

In 1983 Rome Air Development Center (RADC) published "Report on a Knowledge-Based Software Assistant" [3]. This document brought together key ideas on how artificial intelligence (AI) could be used in the software development process. Since then RADC has embarked on the first of three contract iterations to develop both a Knowledge-Based Software Assistant (KBSA) and the enabling supporting technologies. This paper will describe: 1) History leading up to KBSA, 2) What is KBSA, 3) Development strategy and current status of KBSA, and 4) Concluding remarks.

HISTORY

The KBSA research program is a natural progression of research and development undertaken by RADC in its continuing pursuit of a solution to the well known "software life cycle problem". This application of AI technology to the problem of software development was a predictable outgrowth of RADC's longstanding commitment to research and development directed at enhancing productivity. From the early 1970's when RADC was championing the cause of "modern" high level languages and "structured" implementation methodologies, a less publicized but important track of research was being addressed on a smaller scale for the development of greater formalism and abstraction for the objects and activities belonging to the technology of software development. The publicity given to the AI community in the late 1970's and early 1980's due to successes in building workable expert systems and the announcement of the Fifth Generation Computer Program of Japan, resulted in the emphasis of AI technology within RADC's research program. This in turn led to the examination of the possibility of applying the technology that worked so well in areas such as geological analysis and locomotive maintenance to the problems of software development and maintenance. This atmosphere, when coupled with the growing compilation of results from earlier research, encouraged the selection of software development as an

application to drive and demonstrate AI technology research developments. The particular paradigm selected and eventually identified as the KBSA was the result of careful consideration of the state of technology as demonstrated by prior work, and the goals and practical requirements of a system to support software development.

Throughout the 1970's, concurrent with the major RADC projects in language/compiler systems and programming methodologies, efforts were undertaken which explored formalisms with which to better describe the objects and algorithms comprising software. It was recognized that there were many flaws with the existing manner in which programs were created and the languages in which they were expressed. A few of the outstanding problems that were addressed during this time included: programs could be syntactically correct without providing the desired solution or being logically correct; and, even with "better" high level languages, programs were incomprehensible, even to the author with the passing of time. Each of these problems and the research efforts addressing them had a part in exposing the members of the Command and Control Software Technology Division to the work being performed in the world of AI, and while simultaneously providing necessary support for some of the important AI research ideas of the time.

The problem of logical correctness of programs was attacked in many ways. However, the two that are important to the KBSA (even though not receiving widespread adoption in the world of software engineering) are formal proofs of correctness and formal specification languages. In late 1974 an initial effort was undertaken to explore the applicability of formal verification methods to existing programming languages. This process of "verifying" a program's correctness consists of establishing by mathematical proof that whenever a program is executed with specified input data and execution environment the execution will terminate and upon termination the values of program variables will meet output specifications. The foundations of formal program verification are identical with those of a significant body of the work in automatic programming. As might be expected, many of the same individuals are involved in both areas of research. The need for formal specification languages was emphasized by the difficulty of verification of programs in existing computer languages. In 1976, research was initiated to develop a "language" that could be used to provide formal descriptions of programs. The specification languages resulting from this research were found to be unwieldy for extensive use by humans and as is the case with formal program verification technology are not known to have achieved widespread use. However, formal specifications are particularly important to the KBSA because they provide the formalism needed to enable

reasoning about programs. Additionally, these particular research efforts caused RADC to become involved in a progression of efforts addressing automatic programming which continue today.

The need for a more natural and abstract method of expressing a problem solution to a computer led to the exploration of the concept of a Very High Level Language (VHLL). The goal of this research was to provide a language system combining the capabilities of conventional languages with those of logic programming systems enabling the user to program not only computational processes in the conventional sense, but also "reasoning" processes. From this research at Syracuse has emerged an evolving family of languages which exhibit many of the characteristics that will be needed in the Wide Spectrum Language (WSL) of the KBSA. This research was facilitated by the early theoretical work of Robinson [14] which has provided much of the foundation for logic programming. Through this work at Syracuse University, which began in the mid 1970's, RADC has been cognizant of the potential for a software development paradigm unlike that existing for conventional programming languages.

The arrival of practical diagnostic systems based on AI technology in the late 1970's led to a project to investigate the possibility of creating a knowledge-based system that would diagnose software systems and assist in their maintenance. The conclusion of this investigation was that this type of diagnostic expert system would be impossible for software because of the inadequacy of the knowledge about software in general and any software system in particular. This initial negative result, along with the dim prospect for immediate relief from automatic programming caused a serious consideration of the alternatives. The alternative perceived to have the greatest promise was that of a knowledge-based system that did not provide total automation of the software synthesis process, but did maintain a total record of all decisions and activities which occurred in the creation of a software system. This system would possess the expertise to automatically perform many of the tedious tasks of program development, but would be guided in the application of transformations by the human user. Communications among members of a development organization would also be enhanced by the monitoring and reporting capabilities provided by the knowledge-based system. These are the concepts that were further developed and described in the 1983 report considered to be the "defining document" of the KBSA.

WHAT IS KBSA?

The KBSA approach is a departure from the existing software engineering paradigm in that it attempts to formalize all activities as well as products of the software life cycle. It is a formalized computer-assisted paradigm for the development, evolution, and long-term maintenance of computer software. KBSA captures the history of system evolution. It provides a corporate memory of: how parts interact, what assumptions were made and why, the rationale behind choices, how requirements are satisfied, and explanation of the development process. KBSA accomplishes this through a collection of integrated dedicated facets and an underlying common framework.

KBSA has four main distinguishing features. First, the specification is incremental, executable, and formal. Incremental means that the specifier may gradually add more detail to the specification and is not forced to initially describe the system in complete detail. Executable means that the specification is "runnable" like a prototype. This allows the specifier to validate the specification against user intent by actually showing him/her the "running" specification. Finally, formal means that the specification is expressed in a language with precise semantics, avoiding the ambiguity of natural language.

Second, the implementation is formal, that is, all decisions made during the implementation are captured and justified. Typically, implementation will be done via correctness preserving transformations, thus guaranteeing by default a verified implementation.

Third, project management policies will be formally stated and enforced by KBSA. That is, project policy will define the relationship between various software development objects (eg. requirements, specifications, code, test cases, bug reports, etc.) and then be enforced by KBSA throughout the software development process.

Fourth, and finally, maintenance will be done at the requirements and specification level, rather than via patches to the code. That is, since maintenance activities are normally a result of new or better defined user requirements, it makes sense to reflect this in the requirement/specification.

In order to build a KBSA, the authors of the initial report point to the need for specific supporting technologies. These supporting technologies fall into

four main categories: a wide spectrum language, general inferential systems, domain specific inferential systems, and system integration.

A wide spectrum language (WSL) is a single language which provides the user with the ability to capture the formal semantics of the system under development regardless of the level of detail (or the step in the development cycle). A wide spectrum language is both a language and an environment. It must provide uniform expressibility, regardless of what is being described (ie. requirements, specifications, code, test cases, project management policy, etc). Not only must a WSL be able to express these objects, it must do so in a way which is consistent at all levels, both syntactically and semantically.

A general inferential system is a system which supports reasoning. In particular, we are concerned with the overall efficiency of this reasoning, how to capture such things as logic in inference rules and data structures, the quality of explanation generated by the system, and the ability to apply this inferencing power to specific domains.

Domain specific inferential systems extend general inferential systems to include aspects unique to software development. This topic focuses on the knowledge representation of software development objects and inference rules and, in particular, how they can be formally represented and used for further reasoning.

System integration deals with the inherent competition between facets and how a technology base can be put together such that all phases in the software development process are supported sufficiently.

DEVELOPMENT STRATEGY AND CURRENT STATUS OF KBSA

When the KBSA report first came out, it was clear that the supporting technologies were not adequately developed. To address this shortfall a KBSA was to be developed in three iterations. The first iteration was aimed at designing the individual facets and seeing where the supporting technologies could be pulled along. Additionally, there was a desire for an advancement of the understanding of the software development process, particularly within this new KBSA development paradigm. In line with this concept, work began on a Framework (FW) and five (5) facets:

Project Management Assistant (PMA), Requirements Assistant (RA), Specification Assistant (SA), Performance Assistant (PA), and Development Assistant (DA) [DA will not begin until FY 88]. Though boundaries between facets may appear in the first iteration, they will blur in the second iteration and disappear completely in the third iteration.

First Iteration

The results of this have been twofold. First, each facet has pulled at the supporting technologies such that they have advanced the state of these technologies. Universal solutions were not sought, rather solutions unique for each facet have been found. Secondly, each facet developer has made progress in formalizing their particular life cycle phase. This formalization has focused both on the products of individual phases (eg. requirements, specification, and code), but more importantly on the process of how these products came about. In this section the basic approach and milestones of each contractor will be described. Included in this description will be the formalization of the particular life cycle phase and the impact on supporting technologies.

Work on the definition of a PMA formalism and construction of a prototype began in 1984 [9]. Kestrel Institute was the developer. The life cycle goals of PMA were to provide knowledge-based help to users and managers in project communication, coordination, and task management.

The capabilities of PMA fall into three categories: project definition, project monitoring, and user interface. Project definition consists of structuring the project into individual tasks and then scheduling and assigning these tasks. Once the project has been decomposed into manageable tasks, it must be monitored. This monitoring is in the form of cost and schedule constraints. Also included in monitoring is the enforcement of specific management policies (eg.DoD-Std-2167, rapid prototyping, KBSA, etc.). In addition, PMA provides a good user interface for project monitoring and project definition. This interaction is in the form of direct queries/updates, Pert Charts, and Gantt Charts.

The above capabilities were important, but would be expected of any project management tool. What sets PMA apart from its predecessors is the expressibility and flexibility of the PMA architecture. Not only does PMA handle user defined tasks, but it also understands their products and the implicit relationships between them (eg. components, tasks, requirements, specification,

source code, test cases, test results, and milestones). Also present in PMA are objects unique to programming-in-the-large (i.e. versions, configurations, derivations, releases, and people).

From a technical perspective, the advances made in PMA include: the formalization of the software development objects enumerated above, the development of a powerful time calculus for representing temporal relationships between software development objects [10], and a mechanism for directly expressing and enforcing project policies.

The work on the Requirements Assistant [2, 15] began in 1985 by Sanders Associates. The main task of RA was to deal with the informal nature of the requirements process. Sanders' intent was to allow the user to enter requirements in any desired order or desired level of detail. It would be the responsibility of RA to: 1) do the necessary bookkeeping to allow user manipulation of requirements and 2) maintain consistency among requirements as they become known.

RA capabilities include support for multiple viewpoints (eg. data flow, control flow, state transition, and functional flow diagrams), management and smart editing tools to organize the requirements, and the ability to support free form annotations to requirements. In addition to this, RA's underlying knowledge representation, Structured Object and Constraint Language Environment (SOCLE), enables RA to identify contradictions and generate explanations.

The main technical thrust has been on how to handle informality when trying to build an underlying representation of the requirements. This is done by supporting incomplete graphical descriptions at the user interface level, but maintaining a consistent, though not necessarily complete, internal representation. This is done via SOCLE that provides a truth maintenance system which supports default reasoning, dependency tracing, and local propagation of constraints. RA provides application specific automatic classification which is used to identify missing requirements by comparing the current requirements against "typical requirements" of a generic system (already represented within RA's knowledge base). This comparison is then used to generate questions of the specifier to either be sure something vital has not been left out or to gather a justification for the difference.

The main goal of the Specification Assistant [1, 8] is to develop a formal specification of the system under development and then to validate it against user intent. The development of the formal specification must be supported in an incremental fashion, modeling the way developers typically construct specifications. The

validation must be done by exposing the specification to the user at the earliest opportunity and continued throughout the construction process. The effort to develop a KBSA Specification Assistant began in 1985. This work is being done at the University of Southern California- Information Sciences Institute (ISI).

SA capabilities include: an incremental specification language which is executable and a natural language paraphraser which will translate a given specification into English. These capabilities have been built on top of ISI's Wide Spectrum Language AP5, and the development environment CLF. SA can currently handle specifications of a couple pages.

The main technical issues concern: 1) identification of specification statements as requirements or goals and the transformation of these from a high level specification into a low level specification and 2) extracting and assembling system views (ie. reusing specifications and parts of specifications).

The distinction that ISI makes between requirements and goals is that requirements are inviolable constraints, while goals describe general behavior which may have exceptional cases not currently covered by the goal. With this distinction in mind SA provides high level editing commands to further transform the specification into a low level specification. Requirements are transformed in a correctness preserving manner to maintain satisfaction of the requirements. Goals, on the other hand, may be "compromised" in order to handle exceptional cases. That is, after a transformation, the meaning of a goal specification may change.

Extracting and assembling system views deals with building up a specification from smaller specifications (ie. reuse of other previously defined specifications) as opposed to the top down refinement presented in the previous paragraph. SA can combine disjoint specifications, but tools are needed to aid in combining specifications which share common terms.

The Performance Assistant [4, 5, 6] work began at Kestrel Institute in 1985 and is expected to run until 1990. Long term goals for a performance facet are to guide software performance decisions at many levels in the software development cycle, from requirements specifications in very high level programs to low level code. The approach is to combine heuristic, symbolic, and statistical approaches which will provide capabilities for: symbolic evaluation, data structure analysis and advice, and algorithm design analysis and advice. This effort is focusing on data structure selection, performance annotations of a specification, analysis and propagation of performance information, and control structure performance analysis.

Technical issues that [illegible] addressed thus far are data structure sel[illegible] using symbolic and heuristic techniques [illegible] development of PERFORMO, a functional specification [illegible] with set theoretic data types. PERFORMO [illegible] to VAL [12], developed at MIT, and SISAL [illegible] at Lawrence Livermore Laboratory. PERFORM [illegible] primarily for DSS work, but is sufficiently expressive to be a good initial specification [illegible] for the next two research issues: subroutine [illegible] and control flow optimization.

The basic strategy [illegible] in DSS is to supply refinement decisions [illegible] are needed by the implementation [illegible] (likely this would be the DA). When a refinement decision [illegible] needed, PA determines the relevant program [illegible] necessary to make a satisfactory selection. The relevant properties would vary on where in the implementation the generator is. Properties refer to how a specific variable will be used and some characteristics of it. These properties could include: whether the variable is random access, ordered, enumerated, [illegible] and/or possibly empty. Based on these properties, specific implementation decisions can be made.

Development Assistant [illegible] most recent facet undertaken, although [illegible] work has not yet begun. RADC will award the DA contract in early FY 88. The basic thrust of this effort will be to derive an implementation from [illegible] completed specification, automating (via automatic transformation) where possible and capturing user-supplied design decisions when needed.

The Framework [7, 11] was considered to be necessary to bring a global perspective to KBSA. Initial work on the FW began at Honeywell Systems and Research Center in early 1986. The goal of the Framework is twofold: 1) to develop an integrated KBSA demonstration and 2) to propose the specification of a KBSA framework through which all facets must interact and communicate. The purpose of the former is to provide a concept definition that would be intuitively obvious to the most casual observer. The purpose of the latter is to facilitate a tightly coupled interaction between facets. The framework will provide a common reference for each facet developer allowing the sharing of information. Interacting with the framework will be a requirement for all second iteration contractors. The result in the future will be a more highly integrated KBSA.

The main technical issues are 1) define minimum functionality which the framework must provide to all facets, 2) define a [illegible] interface to the framework, 3) extend the framework [illegible] distributed environment, 4)

support programming-in-the-large concepts like configuration control, and 5) provide a consistent user interface.

The overall results of the first iteration will be a KBSA concept demonstration consisting of mostly loosely coupled facets with the exception of PMA and the framework which will be tightly coupled. Each facet will exist on separate machines and communicate via the framework. The framework will be responsible for maintaining traceability between software development objects and keeping facets updated. Establishing the initial traceability (eg. the relationship between requirements objects and specification objects) is the responsibility of the involved facets.

In the past, each facet developer has had their own problem domain in which to work. In general these domains have been small or toy-like. For the KBSA demonstration the problem domain will be the air traffic control problem (ATC). This domain has the advantage of being a substantial problem with a variety of real world issues (ie. real time requirements, data base management, user interaction, interaction with the outside world, and changing or not well defined requirements). The intention of the demonstration is not to solve the ATC problem, but rather to have the ATC requirements be a driver for KBSA. The demonstration will most likely focus on some portion of the overall ATC problem.

Second And Third Iterations

The second iteration of KBSA will begin at the completion of the framework effort. All facets in the second iteration will interact with the framework and thus each other. This will be done by either building individual facets in Honeywell's framework, or more likely, individual facet developers will extend their own frameworks (eg. REFINE, AP5, SOCLE, etc) to adhere to the framework specification. Both are acceptable from a RADC perspective. Prospective developers must convince an RADC that either 1) they will use the Honeywell framework or 2) that their framework does or will soon adhere to the standard. There will be a mechanism to allow for some deviations from the framework specification. This is important since at the beginning of the second iteration the framework described in the specification may not be sufficiently powerful to implement all facets or some specific feature of the framework may preclude functionality necessary for a particular facet. For the third iteration there will be no exceptions.

During the second iteration, work will continue on individual life cycle phases, while also focusing on the interaction between facets and the framework.

For the framework contract, work will address raising the functional level of the framework. The goal is for individual facets to be concerned only with activities unique to their respective facets, while the framework will be responsible for all generic tasks (eg. knowledge representation, knowledge base maintenance, communication between facets, policy enforcement, and general inferencing capabilities).

CONCLUSION

In conclusion, there has been progress over the last four years. The question now is, how close are we to a workable KBSA? The answer is greatly dependent upon the framework specification which will come out of this first iteration. If it is sufficiently powerful, we could have a workable KBSA at the end of the second iteration. On the other hand, if most second iteration contractors have to make generic extensions to the framework, we can not expect a workable KBSA until the third iteration.

REFERENCES

[1] Balzer, R. et al., "Knowledge-Based Specification Ass.st ..." Interim Technical Report, RADC, Griffiss AFB, NY, Dec., 1986.

[2] Czuchry, A. J. Jr., "Where's the Intelligence in the Tut... Assistant for Requirements Analysis?", RADC 2nd ... Conference, Utica, NY, Aug 18-20, 1987.

[3] Green, C. et al., " Report on a Knowledge-Based Software Assistant," RADC Tech. Report TR-83-195, RADC, Griffiss AFB, NY, Aug, 1983.

[4] Goldberg, A. and Smith, D., "Towards a Performance Assistant", Interim Technical Report, RADC, Griffiss AFB, NY, Nov., 1986.

[5] Goldberg, A. and Smith, D., "Performance Estimation for a Knowledge-Based Software Assistant", RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.

[6] Goldberg, A., "Technical Issues for Performance Estimation", RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.

[7] Huseth, S. and King, T., "A Common Framework for Knowledge-Based Programming", RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.

[8] Johnson, W. J., "Turning Ideas into Specifications", RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.

[9] Jullig, R., et al., "KBSA-PMA Technical Report", Final Technical Report, RADC, Griffiss, NY, Nov., 1986.

[10] Ladkin, P., "Primitives and Units for Time Specification", AAAI-86, Philadelphia, PA, Aug. 11-15, 1986.

[11] Larson, A. and Huseth, S., "KBSA Common Framework Implementation", RADC, 2nd Annual KBSA Conference, Utica, NY, Aug 18-20, 1987.

[12] Robinson, J. A., "A Machine Oriented Logic Based on the Resolution Principle", Journal of the ACM, Jan., 1965.

[13] McGraw, J. R., "The VAL Language: Description and Analysis", ACM Transactions on Programming Languages and Systems, Jan., 1982.

[14] McGraw, J. R., et. al, "SISAL: Streams and Iteration in a Single Assignment Language", Technical Report M-146, Lawrence Livermore Laboratory, Mar., 1985.

[15] Sanders Associates, "Knowledge Based Requirements Assistant", Interim Technical Report, RADC, Griffiss AFB, NY, Mar., 1986.

A Common Framework for Knowledge-Based Programming

Steve Huseth and Tim King
Honeywell Systems and Research Center
3660 Technology Drive
Minneapolis, Minnesota 55418

Abstract

The advent of personal engineering workstations has brought substantial information processing power to the individual programmer. Advanced tools and environment capabilities supporting the software lifecycle are just beginning to become generally available. However, many of these tools are addressing only part of the software development problem by focusing on rapid construction of self-contained programs by a small group of talented engineers. Additional capabilities are required to support the development of large programming systems where a high degree of coordination and communication is required among large numbers of software engineers, hardware engineers, and managers. A major player in realizing these capabilities is the framework supporting the software development environment. In this paper we discuss our research toward a Knowledge-Based Software Assistant (KBSA) framework. We propose the development of an advanced framework containing a distributed knowledge base that can support the data representation needs of tools, provide environmental support for the formalization and control of the software development process, and offer a highly interactive and consistent user interface.

1 Introduction

The problem however has several positive characteristics. Since the early 1960s, the hazards and pitfalls of large software development efforts have been well documented [Brooks72]. The use of modular designs, information hiding, and chief programmer teams have sought to address this issue and make the problem more tractable. Tightly coupled toolsets, high-level programming languages, and advanced workstations have sought to improve programmer productivity. Yet even with the wide-spread availability of these technologies, the ability to deliver large software systems that are maintainable, extendable, and delivered on time and within budget continues to be an elusive goal.

[Brooks72] describes these problems as the "software tar pit" and cites four kinds of software development categories: programs, programming products, programming systems, and programming systems products. *A program* is software code that is complete in itself, ready to be run by the author on the system on which it was developed. It is the stuff commonly produced in backyard garages by highly talented individuals and is often the object of envy by many software managers when measuring programmer productivity.

The *programming product* has all the characteristics of a program, but it can be run, tested, repaired, and extended by anybody. For a program to become a programming product, it must be written in a generalized fashion, and thoroughly tested and documented so that anyone may use it, fix it, and extend it.

Similarly, a *programming system* is derived from the simple program by integrating several programs into a collection of interacting components that have been coordinated in their common function and created using a disciplined format. Often programming systems must be constructed to be used in an environment other than the one with which they where developed–embedded systems for instance. This requires greater interaction and coordination with other software and hardware components where the target environment has stringent limitations on memory, performance, and basic support services.

This research was supported by RADC contract number F30602-86-C-0074

Finally, Brooks describes a programming systems product as the [illegible] product with the coordination and interaction of a program [illegible].

Each of the categories of software development process differ [illegible] used. A programming environment is usually described as [illegible] cation execution and debugging of software. Examples of [illegible] Interlisp [Teitelman81], Smalltalk [Goldberg84], and [illegible] support for programming products and programming systems [illegible].

In this paper we describe our current research toward a broader [illegible] work. In section 2 we examine the different kinds of [illegible] describes the motivations behind the design of the new [illegible] integrating tools into the framework. Section [illegible] presents [illegible] framework and the organization of our initial prototype.

2 Taxonomy of Software Frameworks

Supporting technologies for software engineering frameworks [illegible] operating systems, database programming languages, [illegible].

Many of the recent successful environments are known as [illegible] grated environment refers to a system that supports [illegible] program and presents a consistent interface to the [illegible] the programmer to perform mental context switches [illegible] ging it. The emphasis remains on providing a [illegible] programmer. Smalltalk and Interlisp are examples of this [illegible].

However, using the terms outlined by Brooks, current [illegible] creation of programs and programming products but [illegible] of programming systems. High degrees of integration [illegible] single programmer and the coordination and communication [illegible] programming projects. Such programming environments tend to require programs [illegible] such as Lisp lists or objects, and to exchange [illegible] intermediate format such as disk files, node names, [illegible].

Support for large programming projects has increased [illegible] Such *large-scale* programming environments are characterized [illegible] grammers with a high degree of communication [illegible] programming requires programming objects to be:

- easily shareable between programmers
- maintained with multiple versions [illegible]
- protected by a user access policy

Unix is an example of a large-scale programming environment exporting a simple [illegible] the individual programmer and the project manager, namely, the composition of small simple programs connected with pipes [Kernighan].

Advances in software specification, construction and integrated environment [illegible] have motivated the emergence of *knowledge-based* programming systems. Software development [illegible] inherently requires the coordination of large amounts of information. Knowledge-based [illegible] capture the large volume of information and coordinate the [illegible].

of the organizational and bookkeeping tasks on *the machine rather than the programmer*. To solve current software development and maintenance problems, a new lifecycle paradigm for the development, evolution and maintenance of large software projects is necessary to produce a fundamental change in the way software is constructed. Maintenance and evolution should occur by modifying the specifications and then rederiving the implementation, rather than attempting to directly modify the optimized implementation. Since the implementation will be rederived for each change, this process must be automated to increase reliability and reduce costs. Basing the new paradigm on the formalization and machine capture of all software decisions allows advanced reasoning mechanisms to assist with these decisions [Green83].

The automated rederivation of the implementation still addresses only the rapid construction of programs and programming products. However, the volume of information and reasoning mechanisms useful for developing specifications can also be used to address the coordination and communication of programming systems. Such knowledge-based techniques provide a high degree of tool integration and support for large projects. The specification of methods, procedures, and communications by large programming teams that occurs during the software development process result in a set of programmable behaviors called a *process program* [Osterweil87]. The process program ensures the coordination between developers, the use of standards, and the integration of cost, schedule, and technical information during the development of the software system. Operations on software objects and necessary *communications between developers* can be automatically initiated rather than requiring the user to explicitly remember to make the necessary requests. This "human in the loop" allows the machine to encourage and enforce the policies currently but only intermittently adhered to by developers.

3 Motivations for the KBSA Framework

Software engineering frameworks for the next generation of programming environments will consist of a knowledge base for containing project data, inference mechanisms that operate on the knowledge base, and a user interface to assist the user in interacting with the environment. To supply these capabilities, four design objectives motivated the development of the KBSA framework: 1) support for the data representation needs of lifecycle tools, 2) support for the coordination of activities that occurs during the software development process, 3) support for multiple levels of integration of tools, and 4) a highly interactive and consistent user interface.

3.1 Representation of Programming Knowledge

Current operating systems define a set of prescribed file attributes used by the applications they support: file access data, last written date, file owner, etc. Only the file contents and the file name may be generally changed by an application programmer. Access to the file object is limited to references to the system-defined attributes usually with a cost of additional system overhead or the use of some unenforced file-naming scheme to categorize data. The framework should permit applications to include any number of attributes to the data object and allow the data object to be accessed associatively by any one or more of the attributes assigned without excessive system overhead.

The availability of such a database also provides an efficient mechanism for intertool communication and control [Balzer86]. Characteristic of certain current programming environments is the ability to join together two tools such that as information is produced by one tool, it is immediately passed to the next. This capability, referred to as "piping", permits rapid construction of new tools out of existing tool parts [Kernighan81].

Although highly successful, such environments suffer from significant performance limitations due to two major structural flaws: the passing of character strings through pipes is inefficient and the use of dataflow

as a weak
fram manage the
chara they share
such use and
of spec versity of the data

Th wi support
In a emphasized
ma overall
and instead the fra
to co of fun
curr of the inform
of the

Fun of a large softwar
mo inseparable from inter
ma greatly perform
to the program
pro on the
d examining
p the data that
would

coordination:

de pment of
project data
widely available softw
user and pencil based
edures. The ability of new
ed by the communi
ta ties of softwar
processes that
ported

engineering environment
the lifecy
that is

the process def
definition is
value of an engineering
inflexible proce
the process. The
understanding of the

a static system but
support of
ption of all
information
permanently
policy information
ent of a poli

and for characterizing software products, schedules, and users, and clausal logic for describing assertions and for reasoning about the objects.

The concept of programming is being applied to itself [Osterweil87]. As a program is a description an intended behavior that is to be executed by a computer, so should a process program describe the software development process that is to be executed by a software development organization. The assumption is being made that if the process can be formally described then it can be automated and consistently adhered to. By formalizing the methods that are currently being used to develop software, we will be able to correct deficiencies, and incrementally enhance the way software is constructed.

3.3 Integration Strategies

The development of software engineering environments that make use of current tools and are extensible to new tools will depend heavily on the integration strategy used by the environment. Integration refers to the ability of the framework to control and coordinate the operations performed by the tools and to ensure consistency of the data that they manipulate. Many relationships can be defined among tools that have not been designed to share data. Relationships exist among cost data managed by a cost tool, scheduling information managed by a scheduling tool, and the activities of design, code, and test being performed on the software products by the software tools.

Characteristic of most knowledge-based tools that will be integrated into our framework is the tendency to be embedded in their own local frameworks. The local frameworks support individual forms of data representation and inference mechanisms thus satisfying the specific needs of the tool. As a result, our framework is constructed as a "framework for frameworks" providing programming system support to local tool frameworks.

The four KBSA facet efforts that have been completed or are in progress are supported by three different local frameworks. Socle [Harris86], which supports the Requirements Assistant, is a frames-based system that has been extended with mathematical constraints. The mathematical constraints provide a "triggering" capability to continually reestablish consistency of the knowledge base, giving an electronic spreadsheet effect. Whenever one of the inputs is changed, the computation is reevaluated so that the effects of the change are fully propagated.

The Specification Assistant is based on the Common Lisp Framework (CLF) [CLF85]. CLF provides programming environment primitives (modules and workspaces) used to represent programming knowledge. The system is based on the very high level language AP5, an extension of Lisp, that provides a relational representation of the knowledge that can be accessed and described using first-order logic formulas.

Finally, *Refine*™ [Refine86] is used to support both the Project Management Assistant and the Performance Assistant. Refine is the most advanced framework from a production quality perspective and is an integrated language-programming system along the lines of Smalltalk and CommonLisp. The environment is designed to be a manipulator of the language in which it is constructed. The language merges a functional language containing Lisp-like characteristics, an object-oriented capability, and a first-order logic capability. The system is focused on rapid construction of Refine programs by the single user.

From this list of frameworks, we believe several common characteristics can be identified. Each system supports an object-oriented or frames-based view of programming knowledge that has been augmented with inference mechanisms for making logical assertions about the knowledge. By removing overlapping and conflicting capabilities, a set of "common denominator" capabilities that will be initially supported by a common framework can be defined. This list will provide similar capabilities to what each facet expects from its own framework. As new tools and capabilities are defined, the our framework must be able to be modified and extended to meet these new requirements.

3.3.1 Levels of Tool Integration

Integration in current programming environments is generally restricted to the sharing of user-defined data objects out of a small set of framework-defined primitive objects. These primitive objects are defined by the framework or operating system and usually consist of simple byte-oriented files. Higher level program objects such as tables, trees, and abstract syntax graphs must be built out of primitive objects in order for the tools to communicate with the framework. Each tool must make use of the same set of mapping routines referred to as translators in order to share the same high-level object.

The following description highlights the principal levels of integration that are possible.

Minimal Integration Figure 1a shows integration where the framework provides little support for tool integration. Each tool is loaded with its local framework and input data and then executed. The input objects and any additional output objects are returned to the framework when the tool completes. The framework is not able to manipulate the contents of the objects. This approach is similar to current practice where tools exchange data through files. The smallest data unit provided for use in team programming is a file.

Integrated Tool Data Objects This approach allows the framework to manipulate the data objects used by the tools and other tools to manipulate the internal representations of the objects. Each tool is loaded with its local framework. However, tool data is stored by the framework in the form of objects that can be accessed and managed by the common framework. Tool data objects are translated into and out of the internal tool data formats.

Standardized Framework Services The tool is reconstructed to use the common framework services shown in Figure 1c. The tool will be tightly integrated and will start to assume the role of a set of loosely coupled functions rather than a single "black box" operation. The tool increases use of the common framework inference mechanisms rather than its original framework. The original framework is now completely eliminated thus removing the need for expensive import/export operations.

Standardized Object Representations Once the tools have been integrated to operate on objects, the objects they manipulate can be standardized as shown in Figure 1d. This eliminates the need for translators to convert from one tool knowledge representation to another.

A toolset for an environment can never be complete. New tools will be developed and old tools will be modified and customized for particular problem domains. It is important that all three integration levels exist at each of the integration levels described. A tool will gain acceptance and is modified and customized it will progress down the hierarchy to benefit from tighter integration with the other tools.

3.4 Consistent User Interface

In any system with which a user must interact to effectively accomplish useful work, the user interface plays a crucial role. Because of the range and amount of information available, the variety of user sophistication, a powerful and flexible user interface will be critical. Currently, many techniques are applied to constructing a working user interface. This section describes the principal techniques we plan to include in the future.

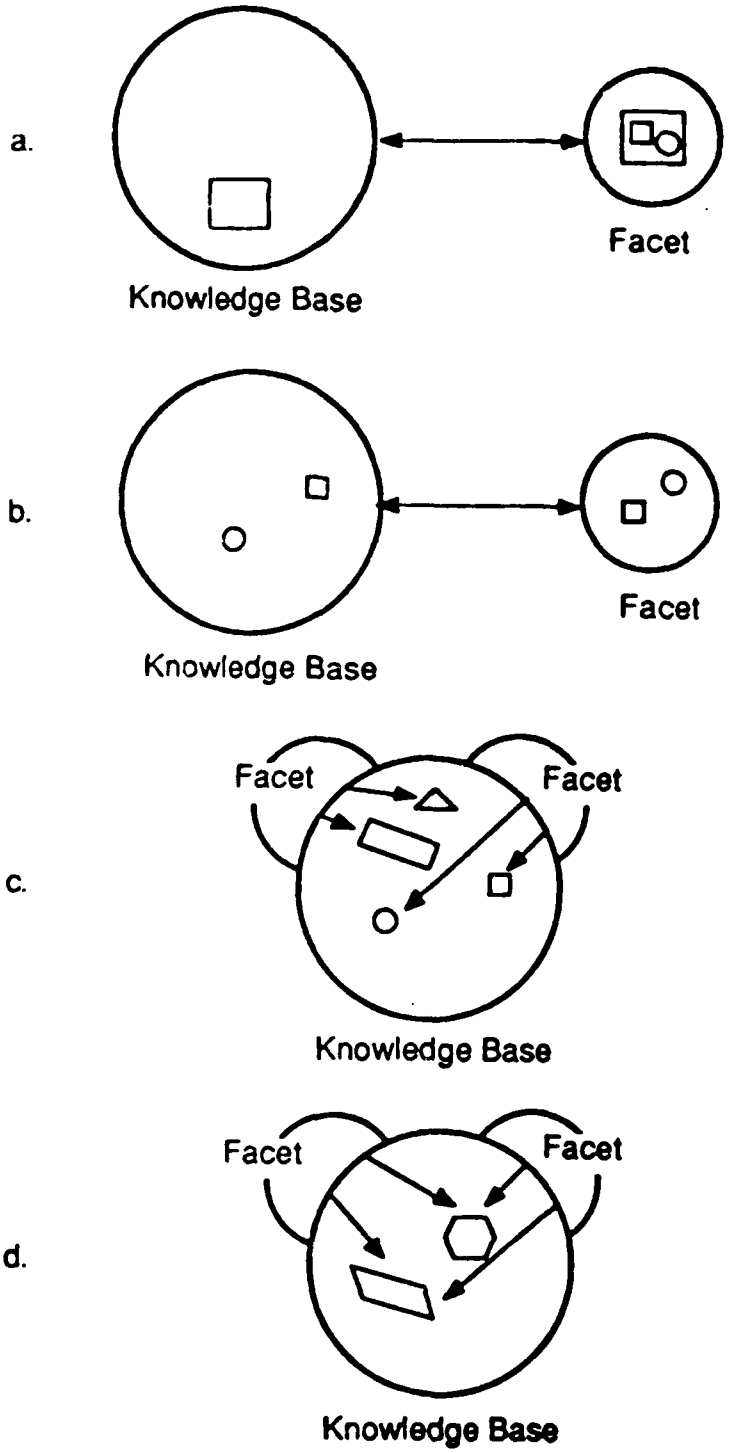


Figure 1: Integration Strategies

The knowledge base will contain project management information such as schedules, [illegible] resources, resource allocation, costing data, project structure, and policy. It will contain technical information such as requirements, specifications, code, test data, and test results. Users will include technical experts to technically naive management, and everyone in between. A useful software development environment must accommodate many kinds of user requests for a variety of different kinds of information and assistance.

We have identified several concepts that are fundamental to user interfaces. First, the [illegible] of interaction must be consistent. The same mechanism should be used to perform the same operation in different context. For example, it is undesireable for a requirements editor to be invoked via a command line interface while the specification editor is invoked via mouse selection.

Help and query facilities that provide accurate, up-to-date information must be accessible using a consistent interface. A help facility should be available to the user at any time and within any context using a common mechanism for requesting help. Exit from the help processor should return the user to the original state.

Second, the interface should be friendly [Meads85]. That is, it should provide unobtrusive access to the system which puts the user at ease. A friendly interface is: cooperative, forgiving, and conducive. A cooperative interface actively assists users by keeping them aware of basic capabilities, encouraging exploration of additional capabilities, and keeping the user informed on what the system is doing and what progress is being made. A cooperative interface will allow the user to suspend a task and quickly return to it at some later time without loosing context or information. A forgiving interface is designed to expect mistakes by automatically correcting simple errors and allowing the user to easily recover from blunders. Finally, a conducive interface is reliable, predictable, and encourages the user to explore new capabilities without fear of destroying vital information.

Third, the interface should provide access to intelligent tutoring and query facilities [Sondheimer82]. Studies have shown that such on-line assistance can substantially affect a system's effectiveness, usability, and learnability. Users require easy access to information that describe features and functionality of tools as well as to data in the knowledge base such as available resources, schedules, and task assignments.

Finally, since users will have diverse backgrounds and varying degrees of technical knowledge, the interface should have some mechanism for user modeling [Wahlster86]. This includes determining and recording beliefs, goals, plans, and knowledge characteristics of the user in an attempt to tailor the system's interaction. In doing so, the system can more appropriately determine what kinds of information to supply and actively assist the user in achieving goals.

Many approaches can be taken to satisfy these requirements for a good user interface. [illegible] suggests seven factors to consider in determining how to provide required functionality:

1. Cost of the interface. Different kinds of interfaces (menu, command-line, graphics, natural language, etc.) require differing amounts of effort in implementation and maintenance. [illegible] balanced against the required functionality.
2. Ease of learning. Some interfaces (menu, natural language) are easier to [illegible]. The interface complexity must be geared toward the user community.
3. Conciseness. Some interfaces are more concise and require less interaction to [illegible]. The conciseness (verbosity) of the interface must correlate with the tasks to be [illegible].
4. Need for precision. Some interfaces are less precise (natural language) than [illegible]. The interface must match the precision required by the task.
5. Need for pictures. In some cases, pictures communicate information much [illegible]. The interface should be capable of displaying pictures whenever appropriate.
6. Semantic complexity. The required complexity of the interface is directly [illegible] plexity of the task being performed. The interface must balance its own [illegible] the tasks it will be used in.

7. Promising more than can be delivered. In most cases, if a system has a complex interface, users will expect complex behavior from the underlying system. The interface must not mislead users by shrouding a simple system in a complex interface.

4 Architecture Overview

Figure 2 describes the overall architecture of our framework. Each tool and its knowledge base are used by the programmer on separate Common Lisp platforms called "Lisp worlds." A Lisp world is a Lisp virtual address space where Common Lisp applications may execute but may not directly interact with other Lisp worlds. The KBSA framework is responsible for all communication between the Lisp worlds.

When a tool is invoked, it is loaded into a Lisp world and run. Tools having their own framework will be loaded with an initial context prior to execution. Conversely, when the operation has finished, the local Lisp context is restored to the KBSA knowledge base to enable sharing the information. The tool will optionally remain loaded in the Lisp world to facilitate subsequent executions.

These capabilities are supported by two KBSA framework components: knowledge base and client handler. Each of these components are described further below.

4.1 Knowledge Base

The knowledge base manages the "corporate memory" for the programming environment including tools, project data, and project policy. It is constructed on a single Lisp world. All objects managed by the knowledge base are defined and manipulated using four types of constructs: object schema, methods, rules, and demons. Object instances defined by the schema may be distributed across multiple workstations, be associatively accessed using any combination of their attributes, conform to an access control policy, and contain a derivation history of all changes made to the object. The knowledge base will enforce project policies and data integrity relationship on all object instances using the rules and object constraints described to it

The four construct types provide the characteristics of object-oriented programming together with a logic system for declaratively describing relationships between objects and inferring results. Each of the constructs are defined further.

Object Schema and Methods Objects and methods in the knowledge base support abstractions characteristic of object-oriented programming environments [Goldberg84] that have become popular. This concept has been extended by adding additional characteristic of logical assertions.

Each slot in an object schema can be annotated with logical predicates, called slot assertions, which formally describe either the contents of the slot, or a function for computing its value. These predicates and functions are used to establish policies about object classes and ensure integrity of the data.

Object schema define an object class containing a name, optional supertypes, and a list of slots Methods define the messages accepted by the object class and the operations that manipulate its instances.

Demons A demon associates an object method with a predicate condition. The method is invoked whenever the predicate is satisfied. The bound variables to the logic query are used as parameters to the method.

When a demon is triggered, the method is executed in the Lisp world whose operation on the knowledge base resulted in the demon predicate being satisfied. Demons are a primary mechanism used describe program management policy

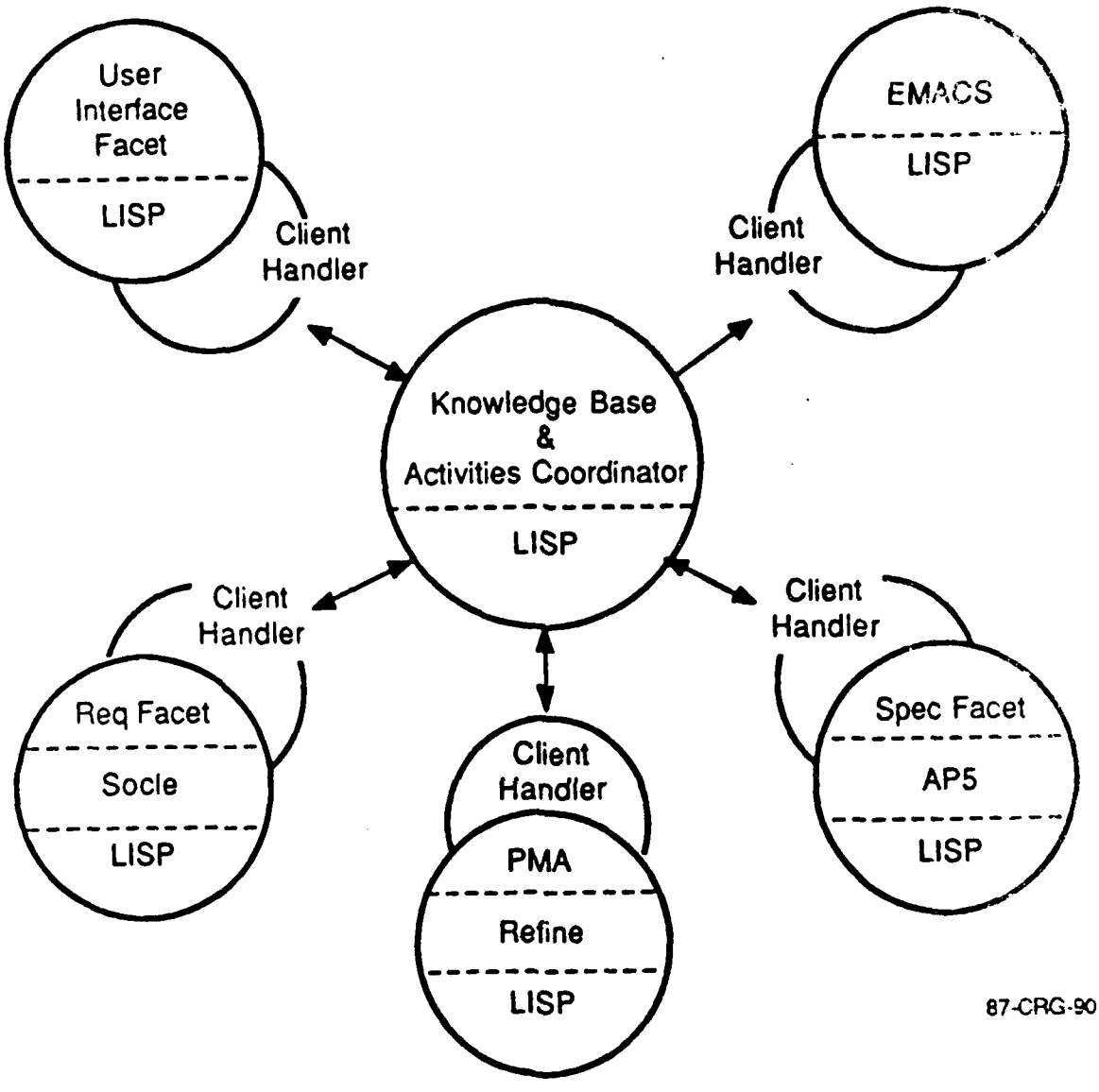


Figure 2: KBSA Framework Architecture

Rules Rules describe logical predicates used to infer relationships in the knowledge base. They will be used in both slot assertions and demons.

4.2 Client Handler

The Client Handler manages the context of a single user Lisp world. It receives messages sent from the knowledge base to load data objects into the Lisp world and to execute object methods. It is also responsible for providing a transparent interface between tools and the central knowledge base. The client handler is the network manager for a single node and ensures that the data on the local node and the knowledge base remain consistent.

4.3 Implementation Details

A prototype with some of the characteristics of the system described has been constructed to provide a testbed for experimenting with framework concepts. It is hosted on a network of Symbolics workstations and makes use of the Symbolics networking protocols. The following section presents the details of the implementation.

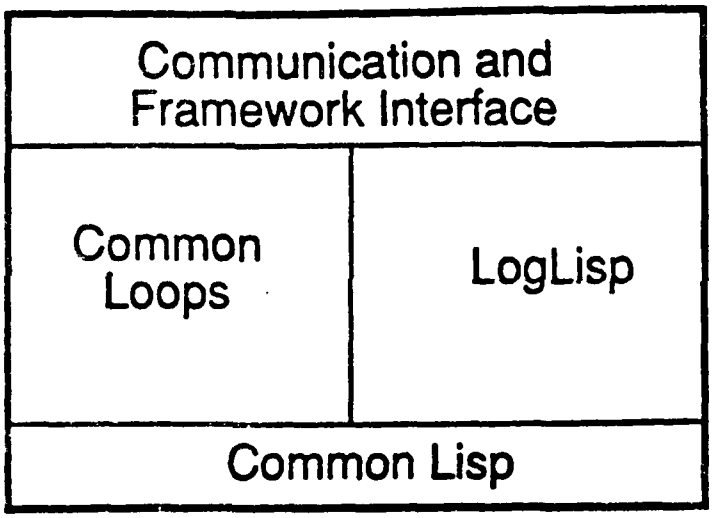


Figure 3: Framework Internal Organization 87-CRV-1082

The object-oriented and predicate logic capability is provided by unifying the LogLisp [Carciofini86] language with CommonLoops [Bobrow85] as shown in figure 3. CommonLoops is a set of object-oriented extensions to Common Lisp. LogLisp is a language which introduces a Horn clause theorem prover into Lisp. As object classes and instances are created, predicates are asserted into LogLisp to map the unification semantics onto the object slots and values.

Both slot assertions and demon objects specify LogLisp queries used to determine slot values and invoke methods respectively. Each modification or creation of an object results in the list of predicates being evaluated. The associated method is invoked or slot value assigned for those predicates returning values. Index tables are constructed to assist the knowledge base minimizing the number of predicates evaluated for each knowledge base operation.

LogLisp also provides a powerful associative access mechanism to any object in the knowledge base by simply

formulating a query containing the attributes of the desired object. This enables [illegible] constructing complex access structures to the objects of interest and instead manipulate [illegible] their attributes.

5 Conclusion

We have described the requirements and motivations of a common KBSA framework [illegible] services not available in the tool frameworks. An initial architecture for such a system [illegible] well as its implementation. The construction of advanced software engineering frameworks [illegible] ily from many other developing technologies of large databases, distributed computer systems [illegible] languages, and natural language understanding. In our investigation of frameworks we have [illegible] examine a broad spectrum of relevant topics without falling into any of the technological [illegible]

Thus, several issues remain to be adequately examined. Many of these are [illegible] and include locking and serializing knowledge base transactions, improving system [illegible] our integration approaches with actual knowledge-based tools. We ascribe to an [illegible] develop solutions to these issues and therefore will continue to expand the prototype.

The real success of the approach however, will be determined by the ability to [illegible] minimal set of software development tools into the framework and begin using the environment to develop itself. To achieve this goal, we have begun to integrate an internally available expert system shell and other Lisp based tools into the framework. This has been particularily useful in continually pulling our focus back to reality where performance and manipulation of large amounts of information is necessary.

References

[Balzer86] R. Balzer, "Living in the Next Generation Operating System", Proceedings of the 10th World Computer Congress, IFIP Congress '86, Dublin, Ireland, September 1986.

[Bobrow85] D. Bobrow, et al, "CommonLoops: Mergine Common Lisp and Object-Oriented Programming", Xerox Palo Alto Research Center, August 1985.

[Brooks72] F. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1972.

[Carciofini86] J. Carciofini, T. Colburn, R. Lukat, "LogLisp Programming System Users Guide", Honeywell Systems and Research Center, July 31, 1986.

[CLF85] -, "CLF Overview", CLF Project, USC Information Sciences Institute, November 1985.

[Goldberg84] A Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Ma. 1984.

[Green83] C. Green et. al., "Report on a Knowledge-Based Software Assistant", Rome Air Development Center, June 15, 1983.

[Harris86] D. Harris, "A Hybrid Structured Object and Constraint Representation Language", AAAI-86, Philadelphia, PA, August 11-15, 1986.

[Kernighan81] B. Kernighan, J. Mashey, "The Unix Programming Environment", *Interactive Programming Environments*, eds. Barstow, Shrobe, Sanewall, Mcgraw-Hill.

[Meads85] J. Meads, "Friendly or Frivolous?", Datamation, April 1, 1985, pp. 96 - 100.

[Osterweil87] L. Osterweil, "Software Processes are Software Too", Proceedings of the 9th Software Engineering Conf., Monterey, Calif Mar 30 - Apr 2, 1987.

[Refine86] , -, *Refine User's Guide*, Reasoning Systems, June 15, 1986.

[Rich84] E. Rich, "Natural-Language Interfaces", IEEE Computer, September 1984, pp. 39 - 47

[Sondheimer82] N. Sondheimer and N. Relles, "Human Factors and User Assistance in Interactive Computing Systems: An Introduction", IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-12, No. 2, March/April 1982, pp. 102 - 107.

[Teitelman81] W. Teitelman, L. Masinter, "The Interlisp Programming Environment", *Computer* 14, 4 (April 1981), 25-33.

[Wahlster86] W. Wahlster and A. Kobsa, "Dialogue-Based User Models", Proc. IEEE, Vol. 74, No. 7, July 1986, pp. 948 - 960.

AN OVERVIEW OF THE KNOWLEDGE-BASED REQUIREMENTS ASSISTANT

David R. Harris

Sanders Associates
a Lockheed Corporation
95 Canal Street
Nashua, New Hampshire 03061

The Knowledge-Based Requirements Assistant (KBRA), is an intelligent assistant computer program designed to actively assist engineers in the requirements acquisition[1] and analysis for complex software system. As a component of the the Knowledge-Based Software Assistant (*KBSA*) effort, KBRA addresses the *KBSA* requirements facet goals as outlined in "Report on a Knowledge-based Software Assistant" [Green, Luckham, Balzer, Cheatham, and Rich]. These goals include formalization, knowledge-based editing and management of requirements, requirements review, and requirements testing.

We have found it useful to liken our KBRA to a junior assistant charged with the management of a requirements engineer's notebook. This management entails carefully recording information along with the dependency relationships among requirements statements, generating requirements statements from alternative points of view, critiquing requirements statements, and noting inconsistencies. Creating this behavior in an assistant program has required careful attention to the division of labor between the assistant and the requirements engineer. The assistant knows the vocabulary of requirements work and will, at appropriate times, take the initiative to check for consistency and propagate information. The engineer is always in charge and uses the assistant's input to help make the key decisions.

This paper indicates the KBRA problem definition, describes the KBRA system, very briefly summarizes the knowledge representation approach, and finally points out the relationship of KBRA to related research.

---

[1] Acquisition of requirements, as used here, means the declaration of ideas and their placement within some model or models of the system to be built.

This work has been funded by the Rome Air Development Center at Griffiss Air Force Base in New York under Contract No. F30602-85-C-0267.

DECISIONS FOR KBRA PROBLEM DEFINITION

KBRA is a first step on the road toward machine-mediation of requirements work. There have been many important problems to investigate and we have had to decide how to best allocate resources in addressing these problems. Three major decisions stand out as influential on the character of KBRA. These are the use of realistic DoD examples, the choice of an integrated facilities approach, and the development of a workstation approach to acquisition of requirements.

Realistic Examples

A requirements facet must address programming-in-the-large. DoD projects are often large and overwhelmingly complex. It is crucial that engineers can "get a handle on the problem". In order to ensure that we addressed the really significant problems associated with complexity (and that our results can hence be rapidly assimilated into the DoD community), we have made a strong commitment to using realistic DoD examples. We began the project by developing a protocol, in the form of a working notebook of requirements work on an Air Traffic Control System. This protocol includes requirements for items such as system administrative functions, tracking capabilities and flight plan maintenance. If anything, the study reinforced our concern that efforts to get a handle on requirements problems are exploratory in nature with system description evolving through incremental changes over many iterations. The Air Traffic Control domain was chosen as representative of the complex but conventional software systems which are targeted for development using KBRA.

We have used this protocol extensively in directing our own work. In addition, it has been adopted by the Knowledge-Based Software Assistant Community as its common example problem.

Integrated Facilities

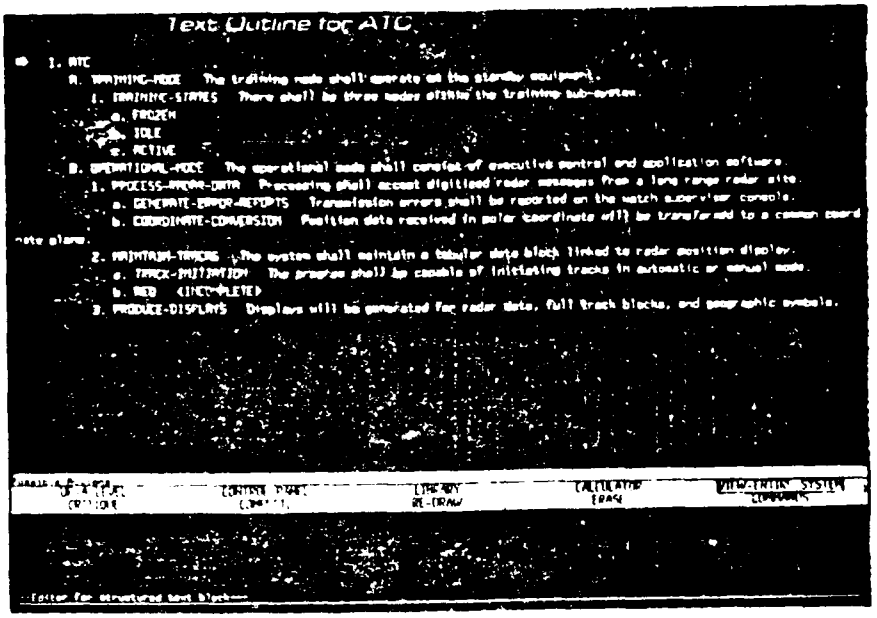
We selected an integrated facilities approach over a prescriptive approach. The facilities that have been developed include 1) capture of informal requirements through an intelligent notepad, 2) capture of dependency relationships using familiar calculator and spreadsheet formats tailored to the relationships between requirements, 3) management of complexity through presentation of familiar system description concepts (data-flow, functional decomposition, state transition), spread sheets, and explanations of requirements entities, 4) knowledge-based editing of these presentations, 5) critiquing of requirements work, and 6) completion of partial descriptions. Examples of some of these will be presented in the descriptions and figures below.

These facilities are integrated through KBRA mediation of the requirements specification process and through integrated reasoning processes. Throughout this process, KBRA checks for consistency, maintains dependency links, and propagates decisions by using an integrated collection of inference processes. KBRA does not enforce a particular methodology. In fact, KBRA does not contain a process model for requirements. We felt that an investigation of a requirements process model was not appropriate at this stage due to the present lack of formalization at the requirements level.

Workstation Acquisition Medium

KBRA provides a "workstation" knowledge acquisition medium. KBRA allows an engineer to enter information in a "natural" (i.e. close to the application) engineering language of graphics and informal notes. The workstation is intended to be used from the first day of a project when engineers are merely sketching out solutions informally and relating associated thoughts.

The acquisition medium approach reflects our recognition that requirements work, much more so than other software engineering practice, is about acquisition. The essence of what requirements engineers do is acquire and apply knowledge about specific domains. By providing a workstation medium, we have been able to build KBRA so that 1) it is accessable by end-users, 2) the results of machine-mediation are immediately evident to these end-users, and 3) feedback from end-users is incorporated to improve the system.



As an illustration of the medium, the figure is an example of the "Intelligent Notepad" acquisition vehicle. The "notes" shown are entered by a user who wished to describe a next generation air traffic control system. When working with the notepad, an engineer can freely enter associated information taking advantage of extensive associative retrieval capabilities that are part of the intelligent assistance provided by KBRA. Associative retrieval refers to KBRA's capability to access information which impinges in some way on a requirements entity. Examples of this would include finding a function which produces a data item, finding text strings that contain a word, or finding an activity that plays a particular role in an evolving system description.

This focus on acquisition and the associated informality of requirements specification are areas in which KBRA is significantly different from more conventional approaches to supporting requirements work.

THE KBRA SYSTEM

The above decisions and the Knowledge-Based Software Assistant metaphor have together guided our work on KBRA. In this section, we will briefly point out the main features of the current KBRA prototype.

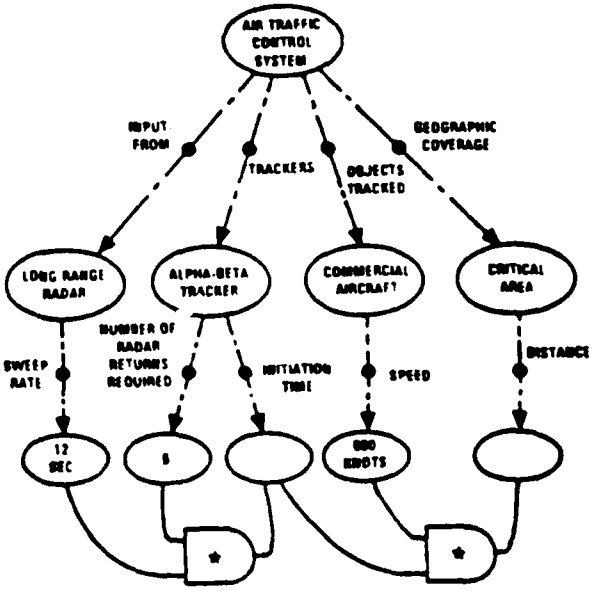
Incremental Formalization

KBRA supports incremental formalization of requirements. In other words, information that is informal (e.g. keywords, unparsed text strings) is stored along with formal requirements statements (decomposition, data flow, event definition, and performance characterization). Thus, KBRA never degrades to a text editor, but rather provides for the editing of all text at the logical level within the context of the evolving system description. This notion has been referred to in [Sterpe] as "structured text".

As work proceeds, KBRA analysis capabilities maintain dependency links, propagate information, and check for consistency as informal is replaced with formal. We will briefly describe each of these analysis capabilities. Dependency links are realized as a form of truth maintenance system. For example, an engineer may state a specific processing time requirement on a system subcomponent in order to cover a near-real-time requirement for the entire system. If KBRA is made aware of this relationship, it will remind an engineer of the connection when a review of the processing time requirement is requested. In addition, KBRA will know that in the event an engineer removes the near real time requirement, the engineer must reconsider the processing time requirement since its support is no longer present.

KBRA propagates information by running rules or procedural attachments when the state of the evolving system description changes. For example, when an external agent, such as a radar system interacting with an air traffic control system, is declared, KBRA automatically adds any known output (such as radar messages) of the agent to the input list for the system being described.

Checks for consistency include looking for data flow anomolies, invoking user-defined constraints on potentially competing requirements, and looking for violation of physical laws. Items which must be specified are often interrelated. When this occurs, KBRA established constraint networks which track dependencies and maintain logical consistency (unit propositional resolution) among the items.

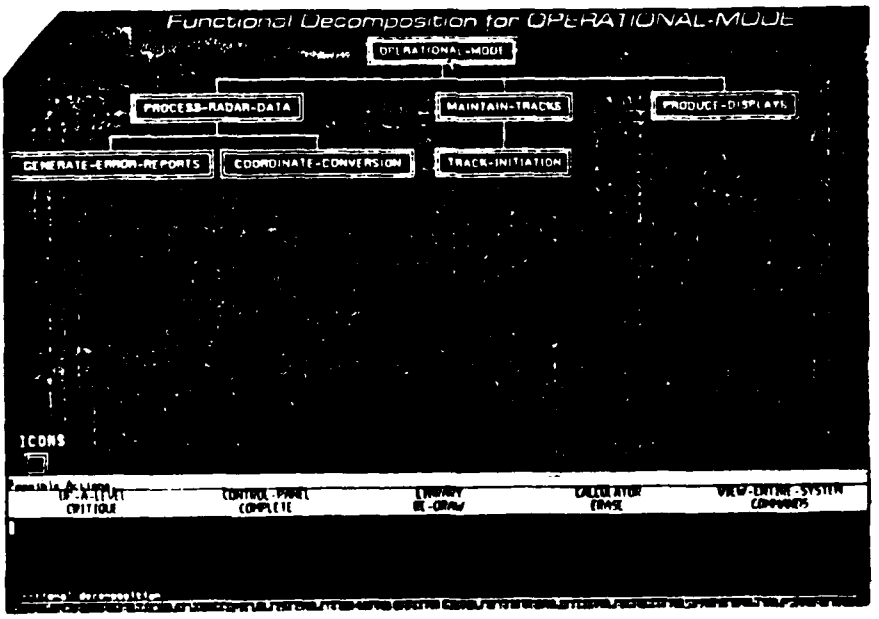


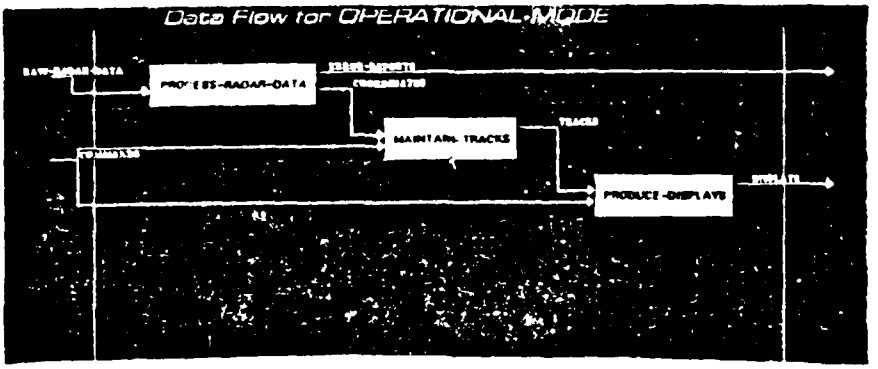
One example of such consistency checking is illustrated in the figure. What is shown is a partial decomposition of an air traffic control system. The sweep rate of the radar, the initiation time of the tracker, and the need to begin tracking airplanes before they enter a controlled area are connected with a simple "distance is rate times time" formula. If KBRA knows of this connection, it will enforce it when an engineer makes exploratiory attempts to specify requirements. In the event that the engineer over-specifies the requirements participating in the formula, KBRA will detect inconsistent requirements and if possible automatically resolve the contradiction.

In addition to assistant behavior, and when appropriate, KBRA takes the initiative and plays a more active role in requirements generation based on its expectations about the system being described. KBRA completes partial descriptions (e.g. completes data-flow), critiques decisions made (e.g. the work is not complete, values chosen are not reasonable in this context, values are inconsistent with other requirements, requirements contain redundancies), incorporates default values and default data flow properties, and automatically classifies objects based on interrelated required properties.

Knowledge-Based Editing of Multiple Viewpoints

Multiple viewpoints are important, certainly for engineers' stylistic choices, but more significantly because software description cannot be completely visualized from any one coherent projection. We have investigated and incorporated standard system description concepts (e.g., data flow, state transition). The development of these concepts has been important knowledge engineering work within the requirements domain. While the methodologies associated with these concepts are frequently inadequate for complete requirements work, the notations used are a particularly useful means of communication between engineers and for helping people to get a handle on problem complexity. A part of our work has been to formalize these concepts in an integrated machine-useable manner. For example, each of the KBRA screen images below displays different information about the operational mode of an air traffic control system. Each in their own way provides a user with a global perspective on the work he or she has done.





Hence, multiple viewpoints are a form of paraphrasing which can help a user manage the complexity. We refer to the viewpoints which can be edited as "presentations". This editing is knowledge-based. In other words, inference based on knowledge about the presentation type and about requirements relationships is used in order to set KBRA expectations, provide feedback, and maintain consistency during the editing process.

Community Knowledge

KBRA embodies and interacts with community knowledge. Community knowledge is domain-specific knowledge about solutions to particular requirements level problems. It can include solutions ranging from decomposition strategies, to typical data, to relationships between performance characteristics. In general it is this knowledge which sets expectations for requirements work and allows KBRA to critique the work that an engineer performs. Except in rare cases, requirements engineers take advantage of the knowledge and experience they have in specifying similar systems. Our perception is that this will continue to be the case in the forseeable future as most new system development will continue to use large segments of conventional components. This suggests that there is a tremendous potential for the reuse of stereotypical solutions. This knowledge can be accessed by engineers and reused to solve new problems. In addition, it allows KBRA to perform analysis on the requirements that an engineer enters.

KNOWLEDGE REPRESENTATION

The highest level view into KBRA reveals two major architectural disciplines and an implementation of those disciplines with a hybrid knowledge representation system. The first architectural discipline is the separation of empty machine from knowledge-base. This is the discipline fundamental to expert system work. Within KBRA it has meant the explicit codification of requirements formalisms such as data, activities, events, states, performance characteristics, other non-functional requirements and the codification of an evolving system description within hierarchies of such objects.

The second architectural discipline is that of a presentation-based interface. That is to say we have taken a systems approach to building an interface which importantly provides multiple viewpoints, associative retrieval, and knowledge-based editing of the evolving system description. This systems approach separates presentation of requirements knowledge (e.g., data flow

diagram, state transition diagram) from representation of requirements knowledge (e.g. a tracker is a kind of air traffic control function which maintains track block records for all aircraft within the airspace). As we mentioned earlier, presentation knowledge is itself an important component of knowledge about requirements. Among other things, KBRA has codified this knowledge, identified commonalities among presentations, and established the connections between presentation and requirements representation. Readers interested in more information about presentation-based interfaces are referred to [Ciccarelli].

Socle (a homonym derived from Structured Object and Constraint Language Environment) [Harris] is the knowledge representational tool which is used for implementing the architectural disciplines. Socle is a hybrid system (with FRL [Roberts and Goldstein] and Constraint-based Languages [Steele] as ancestors). It supports both object-oriented and constraint-based programming styles. The Air Traffic Control diagram illustrated in the earlier figure is an abstraction showing a mix between description through object specification and constraint specification. Within KBRA this information is collected from an engineer through his/her editing of presentations of high level system description (e. g. decomposition, spread sheet relationship) and is propagated to internal representations of the diagram abstraction. Once this information is entered, it is represented as Socle objects. All subsequent inference based on these objects is mediated by Socle.

RELATED EFFORTS

There are several research efforts that are related to KBRA.

The work most closely related to our own is that of the Programmer's Apprentice (PA) [Rich, Schrobe] at MIT. Specifically, a new Requirements Apprentice (RA) [Rich,Waters] beachhead has very recently been started. The RA is like the KBRA in that it attempts to exploit expectations about requirements statements in order to actively assist a requirements analyst. KBRA and RA have different focus however. The KBRA concern for informality and requirements acquisition will probably not be seen in the RA. In addition, the two systems differ in their posture toward machine understanding of requirements. KBRA takes a modest step forward in formalizing this understanding; the RA, as a next generation automatic programming component, will aim at a deeper formalization. Throughout our effort we have worked with members of the PA staff to the mutual benefit of both efforts.

Another related effort is ISI's SAFE [Balzer, Goldman, Wile]. This effort made the significant contribution of identifying problems of informality in requirements specification and of beginning the process of attacking these problems. The SAFE approach is very different from KBRA's approach. Specifically, SAFE emphasizes natural language and total automation of the informal to formal transformation. On the other hand, KBRA emphasizes requirements acquisition through graphical and text modes and the semi-automatic transformation of informal to formal description.

RML [Greenspan] is a language for requirements modeling. In part its inspiration is the SADT model. SADT is a software design methodology that emphasizes dual decomposition of systems from activity and data perspectives. RML supplies the semantics behind an SADT diagram. Its formalization of

requirements constructs (through linkage to first order logic) is an important contribution to machine understanding of requirements analysis.

PAISLey [Zave] is an example of a formal requirements language for a narrow application area, specifically real-time distributed systems. As a narrow domain application generator it is not specifically addressing the goals of *KBSA*. However, in the long term, we anticipate the inclusion of several such application generator capabilities within a requirments facet.

Within the *KBSA* itself, KBRA is related to both the specification facet and the performance facet. The output of KBRA is a set of formal requirements which have been checked for consistency to some level of completeness. These requirements are then handed off to the specification assistant where additional manual and automatic manipulation take place as a formal specification is produced. To a first approximation, KBRA is more about acquisition of requirements and the movement from informal to formal specification. The concerns of the specification assistant are more with verification.

The role of the *KBSA* performance assistant is to select appropriate transformations based on knowledge of performance characteristics associated with choices. This selection can provide efficient solutions at the requirements level as well as the implementation level (where the performance facet prunes the tree of possible implementations as derived from formal specification). For example, cost-reliability trade-offs affect the choice of a tracker algorithm for an air traffic control system. It is important that these trade-offs be considered at the requirements level to ensure that solutions are feasible. KBRA contains such performance analysis capabilities at the requirements level. Included are automatic specialization of algorithms in order to satisfy collections of requirements, and consistency checks between potentially competing requirements on timing, accuracy, or cost.

CONCLUSION

KBRA addresses the machine-management of requirements activities. This is accomplished by providing analysis and presentation support to the process of incrementally developing a formal set of requirements for a complex system. We believe we have taken the necessary first step toward supporting the acquisition of requirements and thus are providing *KBSA* with an ability to handle complex programming-in-the-large systems that are envisioned for the forseeable future.

Future directions, beyond the work we envision for this first prototype,could include 1) development of a requirements process model, 2) building a multi-user environment which addresses issues such as configuration management, partitioning of requirements tasks, and the role of an inteligent assistant as an active agent working with several human agents, and 3) the addition of requirements level validation tools such as general purpose and special purpose rapid prototyping capabilities. In our own efforts, we have conducted a preliminary investigation into using Socle as an executable requirements language for rapid prototyping. Results of this investigation have been very incouraging.

The current KBRA prototype is a demonstration of many of the essential features of the ultimate *KBSA* requirements assistant. It is designed to

support the requirements engineer beginning at the initial phase of requirements work and continuing through to document preparation. In using KBRA, an engineer makes the commitment to working on-line not on-paper. The reward for this commitment is that the decisions made and their associated justifications are not lost, but rather become part of the *KBSA* environment to be available and used throughout the entire software life cycle.

Acknowledgements

Important contributions to the design and implementation of KBRA and to the development of this paper have been made by Chuck Rich of MIT's AI Lab and by J. Terry Ginn, Andy Czuchry, and Patricia Ahern of Sanders Associates.

REFERENCES

Balzer, Goldman, Wile, "Informality in Program Specifications", IEEE Trans. on Software Eng., Vol 4, No. 2, pp. 94-102, March 1978

Ciccarelli, "Presentation Based User Interfaces", MIT AI-TR-794, 1984.

Green, Luckham, Balzer, Cheatham, Rich, "Report on a Knowledge-Based Software Assistant", Rome Air Development Center, Technical Report RADC-TR-83-195, August 1983.

Greenspan, "Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition," PhD Thesis, Dept. of Computer Science, Univ. of Toronto, 1984.

Rich, Shrobe, "Initial Report on a Lisp Programming Apprentice", MIT-AI-TR-354, Dec. 1976.

Rich, Waters, "Toward a Requirements Apprentice. As the Boundary between Informal and Formal Specifications", MIT-AI Memo 907, July 1986.

Roberts, Goldstein, "The FRL Primer", MIT/AI Memo 408, 1977.

Steele, G., "The Definition and Implementation of a Computer Programmming Language Based on Constraints", MIT PhD Thesis, MIT/AI/TR-595, Aug. 1980.

Sterpe, "TEMPEST: A Template Editor for Structured Text", MIT, AI-TR-843, June 1985.

Zave, P., "Executable Requirements for Embedded Systems", 5th Int. Conf. on Software Engineering, 1981.

Overview of the Knowledge-Based Specification Assistant

W. Lewis Johnson
USC/Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292

Abstract

In order for specifications to be used effectively in software development, we need tools to do the following things: a) assist in developing the specification; b) analyze and evaluate the specification; and c) explain the specification. The Knowledge-Based Specification Assistant (KBSA) is designed to provide assistance along these three dimensions. This system aids people in specifying programs, by allowing them to record their initial views of system requirements and to gradually transform them into complete specifications. Symbolic evaluation and simulation tools are provided so that the specifier can run the specification as a prototype, and so that unexpected implications of statements in the specification can be detected. Explanation facilities do retrievals from the specification database in response to user queries, and generate natural-language answers and summaries in response to the queries. An overview of each of these facilities will be presented in this paper.

1. Introduction

Software development and maintenance are complex, time-consuming, and error-prone processes. Software developers must decide what an application is supposed to do, implement the application, and then somehow determine whether or not the implementation really does what was intended. Maintainers must try to understand what the application does, what it is supposed to do, and then decide how to change one or the other or both. At each stage errors and misunderstandings may arise; as the application is revised and maintained, the potential for errors and misunderstandings becomes that much greater.

Research in automatic programming [3, 15] and transformation-based implementation [10, 8, 4] has attempted to reduce the complexity of the software process, by bridging the gap between specification and implementation. Software is developed not by writing code but by writing formal specifications. These specifications are automatically or semi-automatically transformed into executable code. As a result, people can be more confident that the code really implements the specification. Maintenance can be performed by changing the specification as needed, and deriving a new implementation from the modified specification.

The Knowledge-Based Software Assistant effort [13] has embraced the notion of specification-based software development. At the same time, though, it is recognized that automatic programming and transformation-based implementation leave major questions unanswered. First, how is the specification developed in the first place? Specifiers do not just sit down and write a specification; it takes time to work out all of the details of what an application should do. Second, how do we ensure that people really understand what a specification says? It is difficult to wade through the complexity of a typical specification, and recognize where assumptions and design decisions were made. The Knowledge-Based Specification Assistant project attempts to address these questions, by providing aid in developing, evaluating, and understanding specifications.

The Specification Assistant is motivated by the recognition of two key aspects to

specification development:

- specification development is a kind of design process;
- specification development is an evolutionary process.

When specifiers write specifications, they have a set of goals in mind for the specified system to achieve. They attempt to design a specification which determines precisely what the application should do to try to meet those goals. Complexity arises as the specifier identifies exceptions to goals, resolves conflicts between goals, or reformulates goals into new ones that are easier to implement. At the same time, the specifier's conceptual model of the domain and the application constantly evolves; the specifier may discover new goals to be addressed, or change his or her mind about previously stated goals.

The Specification Assistant aids specification development by allowing people to record their initial views of system requirements and helping them to transform this initial descriptions into complete specifications. It also provides specifiers with tools for revising the specification in meaningful ways; the specifier can describe what is wrong with the specification, and the Specification Assistant can then change the specification to correct the flaw. A record is kept of how the specification was constructed and modified; this aids subsequent understanding and maintenance of the specification.

In order to develop a specification properly, a specifier must constantly analyze and evaluate the specification. The specifier must make sure that the specification says what is supposed to say; the specifier must also check if the specification is incomplete or unimplementable, and refine the it as necessary. To assist this process, KBSA provides tools for analyzing and evaluating specifications. The system can symbolically evaluate the specification, or run simulations on test data. It can analyze the specification for various kinds of inconsistency or incompleteness.

KBSA also has facilities for explaining specifications. When fully implemented, these facilities will include showing the specification from different viewpoints and along different dimensions, and paraphrasing the formal specification notation into English.

These facilities are intended as a means for communicating a model of the specified system to the user.

2. Architecture of the Specification Assistant

Figure 2-1 shows the high-level data-flow structure of the Knowledge-Based Specification Assistant, or KBSA.[1] In this diagram, boxes represent data maintained or generated by the system; ellipses represent processes within the system; triangles represent inputs from the user. Everything inside the large box is internal to the Specification Assistant.

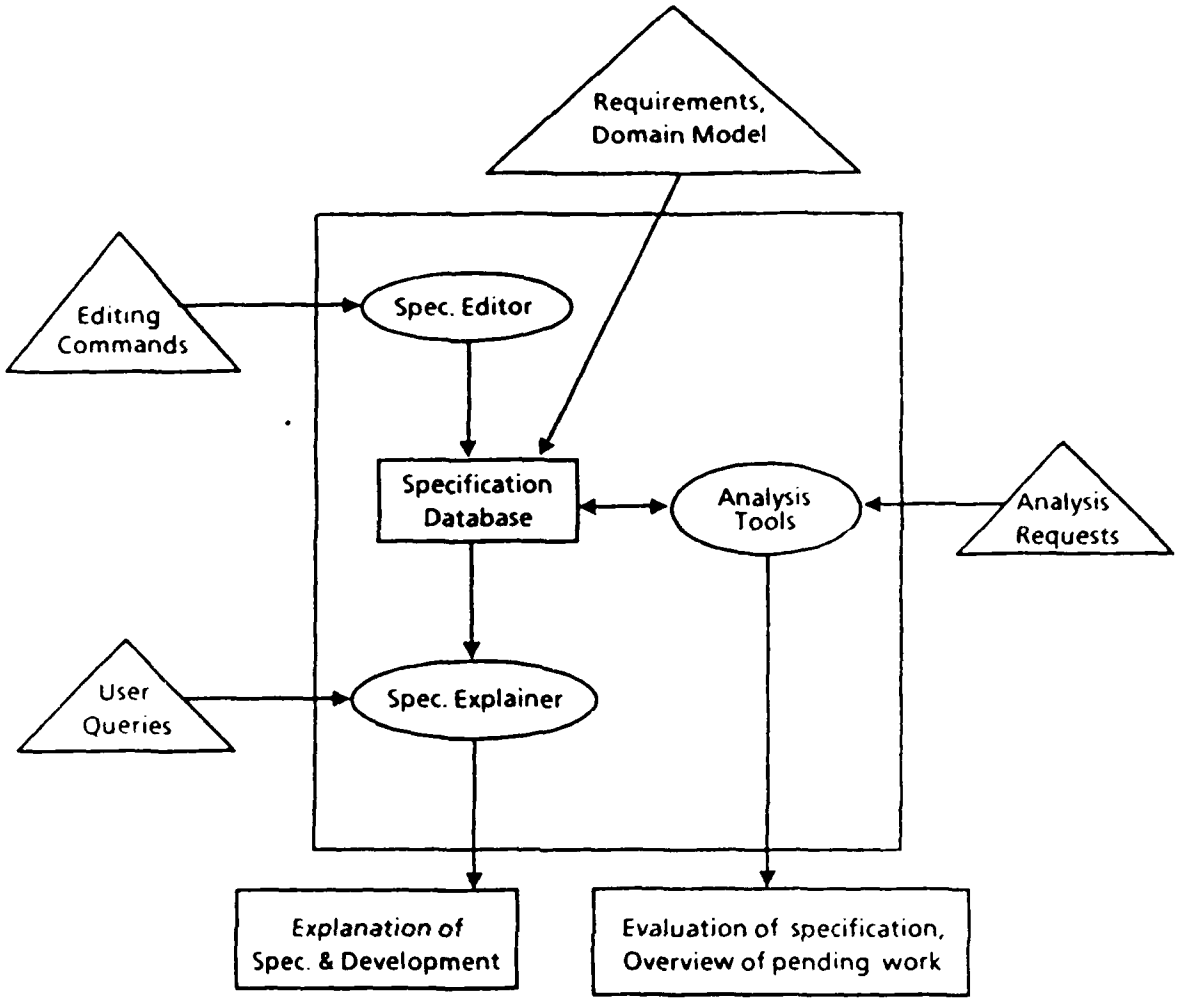
At the start of the process, the user describes the domain in which the specified system will operate and outlines the initial requirements to the system. The domain model consists of gist statements describing objects, relations, and events in the [illegible] requirements are high-level gist descriptions of the desired behavior of the system to be specified, in the form of constraints and sequences of events and states. [illegible] are entered into the Specification Database, a repository of information about specifications that are currently under development, or which have been developed previously.

Currently the requirements and domain model must be entered directly by the user. However, we are in the process of building a translator which can take the domain models used by the Knowledge-Based Requirements Assistant [17] and translate them into Gist. This will allow the Requirements Assistant and the Specification Assistant to share the same model of the domain being specified.

Once the specification statements have been entered into KBSA, they can be gradually refined and revised. These refinements and revisions are accomplished using *high-level editing commands*. High-level editing commands modify the specification in order to accomplish some desired change in the semantics of the specification. A history of the

---

[1] Throughout the rest of this document, we will take "KBSA" to refer to the Knowledge-Based Specification Assistant, not the Knowledge-Based Software Assistant. The ambiguity of the acronym is unfortunate.



Key

Triangles	User input
Boxes	Data
Ellipses	Processes

Figure 2-1: Overview of Specification Assistant

editing steps and their effects are recorded in the specification database.

In order for the specifier to better understand a specification as it is being developed, a set of analysis and evaluation tools are provided. These tools analyze the specification to make sure that it is semantically consistent, and identify places where further

specification development is required. They also provide the means for executing the specification and observing its behavior. The analysis tools can be invoked directly by the user; they are also automatically invoked by other components of KBSA, such as the specification editor, in order to retrieve information about the specification. The analysis tools add information to the specification database, where it is available for inspection by other components of KBSA.

Finally, the Specification Assistant contains a Specification Explainer module. The Specification Explainer's current function is to paraphrase the specification in English, making it easier to understand. However, when users request paraphrases of parts of specifications, they usually have a particular question in mind about the specification. We plan to significantly extend the power of the Specification Explainer so that it can answer general user queries about the specification.

3. Specification Development

We will now look at how specifications are built up incrementally in KBSA. We will examine the following issues:

- high-level editing commands, their purpose and function,
- reusing specifications using specification views, and
- how KBSA can guide the user in selection of editing commands.

At this stage in the implementation of KBSA our repertoire of high-level editing commands are fairly well developed. The view mechanisms and user guidance techniques are at fairly early stages at this point.

We are current working on two applications as test cases for KBSA: a patient-monitoring system and an air-traffic control system. The examples in this paper will focus on the patient-monitoring system. A number of researchers have used this application as an example specification problem. A short description of it, taken from [9], follows.

> A patient-monitoring system is required for a hospital. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance. The program reads these

factors on a periodic basis (specified for each patient) and stores these factors in a data base. For each patient, safe ranges for each factor are specified (e.g., patient X's safe temperature range is 98 to 99.5 degrees Fahrenheit). If a factor falls outside of a patient's safe range, or if an analog device fails, the nurse's station is notified.

3.1. High-Level Editing Commands

To illustrate incremental specification development and high-level editing commands, we will start out with a high-level specification of the patient-monitoring system, and show how this specification is progressively elaborated. This example will go into some detail, so that the reader can see exactly how the process is carried out.

Figure 3-1 shows the high-level Gist specification for the patient-monitoring system. We will not go into the details of Gist syntax here; the reader may refer to Appendix I for a description of Gist. However, to make the specification easier to understand, we include in Figure 3-2 the English paraphrase of this specification generated by KBSA's English Paraphraser, a part of the explanation component of KBSA.

An important point to observe about this specification is that it describes the patient monitor's behavior in a highly idealized, non-operational manner. For example, the `notify-nurse` demon of the monitor notifies the nurse simply by asserting the `knows-unsafe` relation between the nurse and the patient. It performs this action whenever the `safe-signs` relation is false for a patient. Some of the behavior of the application and the environment is underspecified. The relation `safe-signs` currently is described as changing in a random or unspecified way.[2] In the complete specification `safe-signs` will not be random at all, but will be computed from the patient's vital signs. A motivation and analysis of these different kinds of specification idealizations appears in [14]. The specification developer must transform this initial specification into a concrete, operational specification.

The specifier might start out by filling in some of the definitions of the terms in the

---

[2] This information is captured in the definition of the relation as `implicit`, which means that the relation randomly becomes true and false over time, as a consequence of some process that the specification does not describe.

```
{type vital-sign = {'blood-pressure, 'temperature},
 type pvalue = real,
 type day-time,
 type patient with {
   relation reading( vital-sign, pvalue)},
 static singleton type nurse,

 relation medical-entry(monitor, patient, vital-sign, pvalue,
                                                      day-time),
 relation knows-unsafe(nurse, patient),
 implicit relation safe-signs(patient),
 implicit relation time-to-record-signs(patient) ,
 implicit singleton relation current-time(day-time) ,

 static singleton type monitor with {
   demon notify-nurse[ patient]
     when ~safe-signs(patient)
     do insert knows-unsafe(nurse, patient),
   demon record-signs[ patient]
     when time-to-record-signs(patient)
     do parallel loop on{v | vital-sign} do
       insert medical-entry(monitor, patient, v,
                            reading(patient,v,?), current-time(?))}
 }
```

Figure 3-1: A high-level specification of the Patient-Monitoring System

specification, such as **safe-signs**. **safe-signs** is true of a patient if and only if all of the patient's vital signs are within some safe range. The specifier must therefore refine the definition of **safe-signs**, and define any new terms that are introduced in the process. The resulting definitions might look like the following:

```
implicit relation safe-signs(patient) iff
  for all v | vital-sign || safe-vital-sign(patient, v);
implicit relation safe-vital-sign(patient, v|vital-sign) iff
  inside-range(reading(patient, v, ?), safe-range(patient, v, ?));
implicit relation safe-range(patient, vital-sign, range-of-pvalue);
implicit relation inside-range(pvalue, range-of-pvalue);
type range-of-pvalue instances are sequences of pvalue
```

KBSA provides development assistance at this stage of the process primarily by providing structure-editing commands that the user can employ to fill out these definitions. For example, the Add Random Relation command inserts an implicit-

The items in this specification are vital-signs, pvalues, day-times, patients, a nurse and a monitor.

There is only one monitor. The identity of the monitor never changes. Notify-nurse and record-signs are demonic actions of a monitor. The RECORD-SIGNS demon executes a parallel loop when time-to-record-signs is true of a patient. The demon loops over each vital-sign V and inserts a medical-entry consisting of a monitor, a patient, a vital-sign, a pvalue (which is related to the v and a patient by the reading relation) and a day-time (which is the current-time). The NOTIFY-NURSE demon asserts a knows-unsafe relation when safe-signs isn't true of a patient.

There is only one nurse. The identify of the nurse never changes.

There are patients. Reading is an owned relation of the patient. Readings are relations that involve a patient, a vital-sign and pvalue.

There are day-times.

All pvalues are (real) numbers.

There are vital-signs. The only vital-signs are blood-pressure and temperature.

There is only one current-time. The current-time is a predicate over day-times which changes in a random or unspecified way.

Safe-signs is a predicate over patients which changes in a random or unspecified way.

Medical-entries are relations that involve a monitor, a patient, a vital-sign, a pvalue and a day-time.

Knows-unsafes are relations that involve a nurse and a patient.

Figure 3-2: An English paraphrase of the specification in Figure 3-1

relation template into the specification, which the user can then fill out. The Add Definition to Relation command checks to make sure that a relation is declared implicit, and then inserts a user-supplied predicate into the `iff` field of the the definition. The user must do the bulk of the work of defining the terms at this point; KBSA does not attempt to guess the definitions of terms.

Now that the terms in the specification are more fully defined, some initial steps toward operationality can be made. For example, the user may attempt to clarify what data the monitor is allowed to access. The definition of the monitor currently makes use of the **reading** relation, a property of patients; it relates patients, types of vital-signs, and values of vital-signs. The specifier must decide whether the monitor should actually be capable of observing the patients' vital sign readings, and if not how the vital sign readings are to be determined or computed. Also, it is not clear from the specification whether or not the monitor is responsible for determining whether or not `safe-signs` is true. The specification simply states that **safe-signs** is true when the patients' vital signs are within a given range.

We have extended the semantics of Gist to indicate data boundaries. Each type definition implicitly defines a data boundary: any definition inside of the **with** clause of a type is regarded as internal, and everything else is considered external. The user might start by reorganizing the specification to take data boundaries into account. Currently the only definitions internal to the monitor are the demons **notify-nurse** and `record-signs`, describing what the monitor does. We can also move data definitions inside of the monitor definition if we wish. High-level editing commands can be used to perform this task. One command, Make Owned Copy, makes a copy of a definition local to a type, and modifies all references to the definition within the type to refer to the copied definition. We can use this to define a new relation within the monitor, `computed-safe-signs`. The same technique can be employed to other terms that the monitor definition refers to.

The next step is to determine how the monitor is to observe the **reading** relation. This is achieved by introducing a new type called **device**. Devices will have a new

relation, called **device-reading**, whose values correspond to the patients' readings. Every reference that the monitor makes to **reading** should be redirected to refer to **device-reading**. These changes are performed by the high-level editing command Splice, described in [10]. Splice performs the changes listed above; it also defines a mapping between patients and devices, so that the monitor knows which device to read. The resulting definitions of **monitor** and **device** are as follows.

```
type device with {
  singleton relation assoc-patient(patient);

  implicit relation device-reading(vital-sign, pvalue) iff
    reading(assoc-patient(self, ?), vital-sign, pvalue)
  };

invariant for all device ||
  count(any patient || assoc-patient(device, patient)) = 1;

static singleton type monitor with {

  implicit relation computed-safe-signs(patient) iff
    for all v | vital-sign || computed-safe-vital-sign(patient, v);

  implicit relation computed-safe-vital-sign(patient, v|vital-sign)
    iff inside-range(device-reading(assoc-patient(?,patient),v,?),
                     safe-range(patient, v, ?));

  demon notify-nurse[patient]
    when ~computed-safe-signs(patient)
    do insert knows-unsafe(nurse, patient);

  demon record-signs[patient]
    when time-to-record-signs(patient)
    do parallel loop on {v | vital-sign}
      do insert medical-entry(monitor, patient, v,
                  device-reading((assoc-patient(?, patient), v, ?),
                  current-time(?))
  }
```

As we see in this example, high-level editing commands allow specifiers to describe changes to specifications at a more conceptual level. The specifier reasons about what is wrong with a given version of a specification, e.g., a type observes data that should not be observable. He or she then selects a high-level editing commands which implements

the change. The editing command may then perform a number of individual changes to the specification. The specifier does not need to be concerned with how exactly the change is performed. Furthermore, clerical errors are avoided, resulting in a specification that is more likely to be valid.

There are currently over twenty-five high-level editing commands implemented in KBSA; these in turn make use of a number of lower-level commands. We are beginning to reach the point where one can perform extended specification development sessions using KBSA. However, we expect that the editing-command catalog will continue to grow substantially as the development of KBSA continues.

3.2. Views

Although the major thrust of our assistance for specification development is in the area of high-level editing commands, we are also doing work in a number of other areas. One area of research is directed toward breaking away from the linear model of specification development described above. Rather than start with an empty specification and building it up one step at a time, we wish to be able to build a specification by extracting specification components from a common library, and elaborating them in parallel. The following discusses some of the efforts currently underway in this regard.

The KBSA paradigm for specification development involves using a model of the target domain as the starting point for building the specification. One describes the entities in the domain, such as patients and nurses, what their properties are, and what their behavior is. This domain model is used as a basis for describing what the application should do. For example, once we have described what it means for a patient's condition to be regarded as safe, we can use this to define how the patient monitor reacts to unsafe patient conditions. However, there is little that KBSA can do before the domain model is defined. In the above scenario, the user had to supply definitions for relations like `safe-signs`; KBSA's editing commands provide some help in entering the definitions into the specification, but cannot provide any suggestions as to what the definitions should be.

The effort of developing a domain model can be lessened by reusing d[illegible]. Our experience so far has shown that some components of domain mod[illegible] reusable. For example, we find that basic terms such as physical objects, [illegible] people, etc., are used over and over again. We have built a library of such [illegible] and have provided mechanisms for including these into a specification.

Unfortunately, simply building a library of components is not the answer. In any given domain model, each object will be associated with a number of different relationships, constraints, and behaviors. The specifier needs control over which of these relationships and behaviors are included in the specification. Otherwise the complexity of the model will be too great, and the specifier will have difficulty understanding the specification and making sure that it is correct. What is needed, in other words, is a way of constructing a *view* of an existing domain model for inclusion into a new specification.

The patient-monitoring application illustrates with the problems associated with making use of an existing domain model unaltered. Suppose that we started with a standard model of a hospital. Such a model would include information about when patients are admitted and discharged, how monitoring devices in cardiac wards differ from those in orthopedic wards, etc. Such a model would introduce a number of exceptions and special cases into the specification right at the beginning. If a specifier attempts to address all of these aspects of the domain model at once, mistakes are likely to arise. It is crucially important that the domain model be idealized in the initial stages of specification development, so that the initial specification of requirements can also be kept simple and easy to validate. The details of the domain model should then be introduced in a gradual manner, and their implications propagated through the specification.

At the present time KBSA supports a simplified mechanism for incorporating views of other specifications, and of domain models in particular, into a specification. The user can indicate which terms are to be included in a specification. The definitions of those terms are then imported into the specification.

The problem of how to integrate different specifications is in general quite hard. Nevertheless, we have made some strides towards tackling the problem. We have implemented a mechanism similar to Ada packages for inclusion of views into a specification. A specifier can refer to some existing specification, select definitions from that specification, rename them if necessary, and include them in the specification that he is working on. Using this feature, we have started building a library of reusable Gist definitions which can be included in a variety of specifications.

The package technique works fine if two views have disjoint sets of definitions; the situation becomes more complex when the views each define the same terms. The most common situation where this arises is where behavior is described as being random or unconstrained in one view, and deterministic in another view. Such is the case when a view describes the properties of one agent, but only minimally describes the properties of external agents. A view describing what the radar system does might describe aircraft as being in random positions. In this view, the radar system informs the air-traffic control system of the aircraft positions regardless of where the aircraft are. Since the aircraft positions are unconstrained in the radar view, the view can readily be combined with a view in which the aircraft positions are constrained to follow some flight path. We are currently developing analysis tools in KBSA for detecting when unconstrained descriptions and deterministic descriptions can safely be merged.

The next step in this work is to provide better control over how a view is constructed. One typically wants to add or remove constraints from a model, and there are choices to be made as to which constraints should be added or removed. Sometimes constrained behavior is changed to random behavior. In the patient-monitoring example, device readings are a function of patient vital signs. If we were to remove patient vital signs from the model, then the device readings would appear to change randomly. On the other hand, sometimes new constraints must be added when abstracting a view. The general hospital model might describe how patients are admitted and discharged; however, when constructing an ideal view it may be useful at first to ignore admission and discharging and assume that the set of patients is fixed. High-level editing commands will be useful for performing the abstraction steps.

3.3. Providing More Intelligent Assistance

High-level editing commands and view extraction are useful techniques for supporting specification development. However, decisions still have to be made as to which command is the right one to apply. It has become clear that KBSA can perform a more active role in deciding which commands are appropriate. KBSA could assist the user in deciding which commands should be applied to fix a particular flaw in the specification. In addition, it could track the development process, to make sure that the user's intended requirements are properly met in the final specification. Each of these issues are discussed in [14].

We presume that the specifier frequently refines the specification by looking at it, identifying things that are incorrect or missing, decides how they ought to be changed, and then effects the change. We call aspects of the specification that are incorrect or missing *issues*, and intended changes *development goals*. By providing knowledge into KBSA of what kinds of issues to look for, and how issues, goals, and editing commands relate to each other, KBSA can provide guidance as to how the specification can be modified, and perhaps make the changes itself, with user approval.

In the larger scale, KBSA will have the capability of recording and tracking the overall specification development. This involves recording what high-level editing commands were applied, and what goals they were intended to achieve, if known by KBSA. As the process proceeds, KBSA can monitor the development process and check that high-level requirements are not unintentionally violated as the development proceeds.

4. Evaluation Tools

KBSA contains a number of different kinds of evaluation tools, including the following:

- a symbolic evaluator,
- a concrete simulator,
- a number of static analysis tools, including type analysis and resource

analysis.

Each will be described below.

4.1. The Symbolic Evaluator

KBSA's symbolic evaluator is an extension of KOKO, the symbolic evaluator developed earlier at ISI [1, 5, 6]. KOKO was converted from INTERLISP with AP3 [12] to Common LISP with FSD/AP5 [2, 7]. It was adapted to handle KBSA's new internal representation of Gist, reflecting some new constructs that have been added, and changes to the semantics. It was also modified to allow the user to switch easily between different specifications that are being developed simultaneously.

We intend to make several nontrivial extensions to KOKO in order to cover more of Gist. For example, temporal reference and parallel execution are not currently handled. Parallel execution in particular will involve a significant amount of effort. For one thing, KOKO currently has no way of representing nondeterministic interleaving of events. It cannot represent, for example, that three events (patient becomes unsafe, patient becomes safe, and nurse is notified) occur in a specified order but with arbitrary behavior interleaved. We hope that this can be effectively accomplished by allowing concrete sequences of states to be followed or preceded (and thus connected) by indefinite behaviors, which are described by sets of temporal axioms (similar to the axioms in definite states except that time is an explicit parameter).

The major new application of the symbolic evaluator that we envision is in the evaluation of scenarios. That is, the user may present a prototypical sequence of actions, and use the symbolic evaluator to determine whether or not the scenario is a legal behavior, and under what conditions it will be executed. Since people often validate specifications by working through prototypical scenarios, such a feature would be a significant aid for checking specifications.

4.2. The Concrete Simulator

To complement the capabilities of the symbolic evaluator we have also built a concrete simulator for Gist. The simulator takes concrete examples of data and runs them through the specification. Since it cannot be used to execute symbolic data, it is impossible to prove general statements about a specification's behavior using the concrete simulator. On the other hand, the simulator can trace much more complex executions than the symbolic evaluator can. It can execute all of the constructs of Gist, including a number of constructs that the symbolic evaluator cannot handle. It can execute a number of processes in parallel; the symbolic evaluator, in contrast, only follows a single line of control at a time.

4.3. Static Analysis Tools

A number of tools have been implemented, or are being developed, which perform static analyses of specifications. One set of tools, the type checking tools, employ standard techniques for checking the declared types of objects, types of parameters, etc. This form of static analysis is a prerequisite for a number of the tools in KBSA, including both the symbolic evaluator and the concrete simulator.

Further analysis tools will provide feedback on how the specification is constructed, and whether or not it appears complete. One tool under development examines data accesses in the specification, and in particular which accesses cross over data boundaries. If the specifier describes what kind of data accesses are expected, the system will be able to indicate whether or not the specification meets expectations.

5. Explanation

Another major aid to specification understanding is be the KBSA explanation facility. This facility helps users better understand their specifications by producing natural language (English) paraphrases of the specifications, explanations of the behaviors they denote, descriptions of how the specifications were developed, and the rationale behind that development. It extends the capabilities of the Gist Paraphraser [18] which was developed under a previous contract [1]. In the remainder of this section, we first discuss the re-implementation of the Gist Paraphraser and then discuss our plans for

future enhancements to the explanation capabilities of the system.

5.1. The Gist Paraphraser: Current Status

The Gist Paraphraser produces natural language paraphrases of Gist specifications. We originally developed this tool to make Gist more accessable to people who were not familiar with Gist. While the Paraphraser was useful in this capacity, we were surprised to discover that it served another (and perhaps more important) role as a debugging aid. We found that a natural language paraphrase of a specification often made bugs in the specification very apparent that were hidden in the formal specification. Partially, this was due to the fact that natural language is inherently more understandable, but it is also due to the fact that the paraphrase provided the user with another view of his specification that stressed certain aspects of the specification that were not emphasized in the formal notation.

We have re-implemented the old Paraphraser in the new KBSA environment. As with the symbolic evaluator, this involved converting it from Interlisp with AP3 to Common Lisp with FSD/AP5 and dealing with the new Gist syntax. There have been several changes to the paraphraser itself. The major change has been to move from a case grammar based generator to a functional (or unification) grammar based generator. The major advantage of this approach is that it will make the generator much more flexible, which will help us in developing the more sophisticated explanation capabilities we describe below.[3]

To correctly paraphrase a specification, the new Paraphraser (like the old one) needs a few annotations to augment the specification that tell in how certain kinds of constructs (such as action invocations) should be translated. In the old paraphraser, this information had to be hand-coded. One of the enhancements of the new paraphraser is that it can interactively acquire the annotations from the user. Whenever the user has reached a point in the development where paraphrasing could be useful, the necessary

---

[3]A typical problem with unification generators has been their efficiency. By carefully optimizing the code we originally obtained from Columbia University, we have been able to speed the generator up approximately two orders of magnitude so that efficiency is not a problem.

annotations can be added at that point. It can also function without the annotations, at the cost of a somewhat less natural-sounding paraphrase.

5.2. Current Work

We are currently developing a generalized framework for explanation. This framework, called the Specification Explainer, will be used both by the KBSA project and by the Explainable Expert Systems project [16]. The shared framework is appropriate because KBSA and EES are both used to develop software systems in an explainable fashion. EES focuses on the development of expert systems, whereas KBSA is applicable to general software. Nevertheless, many of the same issues apply, such as explaining how a particular term is used in the system, or describing how a given requirement is achieved in the implemented system.

The Specification Explainer will have the ability to plan explanations. The explanation planner will employ a set of explanation strategies in constructing its explanations. Based on what the user needs to know (as indicated, for example, by the queries he poses) the system will plan an explanation to convey the needed information to him. The planner will take into account the user's expertise and prior knowledge to select appropriate ways of conveying information to the user.

The Specification Explainer will allow the user to ask follow-on questions if he doesn't understand the explanations it offers. To produce the follow-on explanations, the system will examine the plan used to produce the explanation that was not understood, and "debug" it by selecting an alternate strategy for conveying the information that was not understood in the original explanation.

The Specification Explainer will also benefit substantially from being able to employ the incremental development history to offer explanations that reflect the system at various stages of development and levels of abstraction. The Explainer will also be able to explain the rationale behind changes made to the specification and specification fragments that are introduced because the incremental development history will record that information.

6. Current Status and Prospectus

In recapitulation, the following is a summary of the steps that have been taken so far in constructing KBSA:

- the basic editing environment for KBSA has been built;
- a library of high-level editing commands have been constructed;
- prototype view construction facilities have been developed;
- mechanisms for describing data boundaries within specifications have been developed;
- the Gist Symbolic Evaluator has been included into KBSA;
- a concrete simulator has been developed;
- static type and resource analyzers have been constructed;
- the Gist Paraphraser has been rewritten and extended.

The following are key issues which we must address in the future in our implementation.

- KBSA should provide more assistance in selecting high-level editing commands. This requires representing more knowledge about what development goals high-level editing commands can achieve, as well as more of the user's expectations about the specification.
- The explanation component of KBSA has not been implementable up to now, because we did not have examples of specification developments to use for explanation. Now that we have those examples, we can start work in earnest of explanation in KBSA.

When these steps are accomplished, and as the existing KBSA components are extended, we expect to see KBSA make significant progress toward being a helpful, cooperative, and intelligent aid to specification development.

7. Acknowledgements

The author wishes to thank the members of the Knowledge-Based Specification Assistant Project for their contributions to the work described here: Robert Balzer, Bill Swartout, Don Cohen, Martin Feather, Jay Myers, Kai Yue, Neil Campbell, Anne Langen, and Ed Ipser. We also wish to thank Lt. Kevin Benner of RADC, as well as John D'Addamio, David Harris, and others at Sanders Associates for providing us information about the air-traffic control problem.

This research is supported by the Air Force Command, Rome Air Development Center under Contract No. F30602-85-C-0221. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of RADC, the U.S. Government, or any person or agency connected with them.

I. A Brief Description of Gist Syntax

This is a brief overview of Gist 1987, the latest version of the Gist specification language. It describes the grammar and semantics of the language, focusing on the differences between this grammar and the one described in the 1980 Gist Manual [11]. Some details of Gist have been skipped here; readers seeking more information should consult the Gist Manual.

The grammatical notation used in this text is an extended version of BNF.

- Keywords in the Gist language are in `typewriter` font. Non-terminal names and BNF notation are in *italics*. All non-terminal names are bracketed by <...>.
- Optional constituents are enclosed in braces: {...}.
- Alternative constituents are separated by the symbol |.
- Constituents may be grouped together using parentheses to indicate the scope of the symbol |: *(...)*.
- The symbol * following a constituent indicates that zero or more instances of that constituent may occur. The symbol + indicates that one or more instances may occur.

Gist specifications consist of a set of declarations. The following kinds of things are declared within Gist specifications:

- types,
- relations,
- demons,
- functions,
- invariants, and
- procedures.

The relationship between declarations in Gist 1987 and declarations in previous versions of Gist are as follows. Types subsume both types and agents in the older

versions of Gist. Relations and demons are just as before. Procedures are what used to be called "actions"; invariants are what used to be called "constraints". Functions, a new kind of declaration, provide a way of defining applicative expressions which are repeatedly referred to in a specification.

I.1. Changes to Expressions and Predicates

We will not go into the details of the various Gist statements, expressions, and predicates here. The meaning of most of them are the same as what was available in earlier versions of Gist. The reader should be aware of the following changes in the syntax of expressions and predicates, however.

- A relation with a question mark (?) in it is used to refer to an object that satisfies the relation. Thus, for example,

  ```
  assigned-ward(nurse1, ?)
  ```

  refers to the object **x** such that the **assigned-ward** relation holds between nurse1 and **x**.

- A double bar (||) in a quantified predicate is used to separate the list of quantified variables from the predicate that is quantified. For example, "all men are mortal" would be rendered in Gist as

- ```
  for all man || mortal(man)
  ```

- The attribute notation of Gist, of the form *<object>:<attribute>*, has been generalized. The constituent following the : can now be a a predicate, function call, or procedure invocation. When this is done, the first parameter to the relation, function, or procedure is elided, as is the case when declarations are embedded (see Section I.2.1) below. For example, the following three expressions are the same:

  ```
  nurse1:assigned-ward
  assigned-ward(nurse1, ?)
  nurse1:assigned-ward(?)
  ```

- The syntax of temporal expressions has changed. Temporal predicates and expressions have the following form:

  (<predicate> | <expression>) **as of** *<time-expression>*,

  where **<time-expression>** can be one of the following:

  when *<predicate>*
  ```

```
(first | last | any | all ) (prior | subsequent ) time
  { || <predicate> }
```

I.2. Type Declarations

Type declarations take the form

```
<qualifier>* type <name> <type-relators>
  { with { <declarations>+ } }
```

The meanings of these fields will be described with reference to the following example type definition:

```
static type nurse
  subtype of person with {
    relation assigned-ward(ward)
    }
```

The qualifiers are shorthand for various common invariants that apply to types, and demons that create and destroy types. They can include any of the following:

- `dynamic` (default) or `static`,
- `singleton`, `non-empty`, `singleton or empty`, or `non-empty or empty` (default),
- `explicit` (default) or `implicit`

`dynamic` indicates that instances of the type may be created and destroyed during the course of a behavior. `static` indicates that creation and destruction are disallowed; the same set of instances of the type must exist throughout all behaviors. `nurse` in the above example is an instance of a static type; the specification presumes that the number of nurses is fixed.

If a type is marked `singleton`, only one instance of the type may exist at any one time. `non-empty` types must have at least one instance at any given time. Note that there is no guarantee that the *same* instance or instances exist at all times; such a guarantee obtains only with static types.

Instances of `explicit` types must be explicitly created and destroyed via `create`

statements and **destroy** statements. **implicit** types, on the other hand, are never explicitly created or destroyed. Instead, declaring a type **implicit** is equivalent semantically to defining a demon that nondeterministically creates and destroys objects of the type, and that is the sole demon capable of creating and destroying these objects. **implicit** types are typically used to define derived types, i.e., where membership of the type is implied by some predicate. For example, the following definition defines the type **patient** as a derived type: a person is a patient if and only if the person has been admitted to a hospital.

```
implicit type patient with {
  invariant (self isa patient) <=>
    ((self isa person) and (admitted(self)))
  }
```

The type relators indicate how the type relates to other types in the type hierarchy. A type can be declared a subtype of another type, a supertype of another type, or can be declared equivalent to another type. Gist now allows sets and sequences of objects to be treated as types; type relators are used to indicate what type the members of the sets or sequences should belong to.

I.2.1. Embedded declarations

Declarations can be embedded within a type declaration. This embedding is a notational convenience; it does not affect the semantics of the definition. When a declaration is embedded within a type it can make use of the symbol **self** to refer to an instance of the type, as in the definition of **patient** shown above. If a relation, procedure, or demon is declared within a type, **self** serves implicitly as the first parameter to the embedded declaration.

The definitions of **nurse** and **patient** shown above are shown in Figure I-1, followed by equivalent definitions without embedding. These examples should make clear how embedded and non-embedded declarations relate to each other.

Embedded forms:

```
static type nurse
  subtype of person with {
    relation assigned-ward(ward)
    }

implicit type patient with {
  invariant (self isa patient) <=>
    ((self isa person) and (admitted(self)))
  }
```

Unembedded forms:

```
static type nurse
  subtype of person;

relation assigned-ward(nurse, ward);

implicit type patient;

invariant for all p|entity ||
  (p isa patient) <=> ((p isa person) and (admitted(p)))
```

Figure I-1: Comparison of Embedded and Unembedded Declarations

I.3. Relation Declarations

Relation declarations have the form

*<qualifier>** relation *<name>* ({ *<parameters>* })
{ (iff | requires) *<predicate>* }

The following is an example of a relation declaration: it states that the relation `unsafe-signs` is true of a patient if and only if **`safe-signs`** is not true of the patient.

```
implicit relation unsafe-signs(p|patient) iff
  ~ safe-signs(p)
```

Qualifiers here have similar meanings to those in type definitions. The `implicit` qualifier in the case of relations is shorthand for a demon that nondeterministically inserts and removes tuples of that relation. Parameters, if present, are in one of the following forms:

- *<type-name>*
- *<name>* | *<type-name>*

In either case, the parameter is constrained to be of type *type-name*.

If the `iff` or `requires` field is defined, it defines constraints on when the relation may hold. `iff` signifies the relation holds if and only if the predicate that follows is true; `requires` signifies that the relation may hold only if the predicate is true, but does not necessarily hold. The predicate may refer to parameters as bound variables; the parameter `p` is referred to in this manner in the definition of `unsafe-signs` shown above.

I.4. Demon Declarations

A demon is an activity which is initiated when some predicate becomes true. Each activity is performed in parallel with other activities in the system, as a separate line of control. Demon declarations have the following form:

```
{ serial | parallel } demon <name> [ { <parameters> } ]
  { let <bindings> in }
  when <predicate> do <statement>
```

An example of a demon is the following:

```
demon notify-nurse[patient]
  when ~ safe-signs(patient)
  do insert knows-unsafe(nurse, patient)
```

The parameter list is used to control how many instantiations of the demon may be active at any one time. Without parameters, the demon is activated exactly once when the predicate becomes true. With parameters, the demon is activated for every distinct binding of the parameters that make the demon become true. As an example, consider the `notify-nurse` demon shown above. `patient` is a parameter to the demon; that means that a separate demon instantiation arises for each patient such that

```
~ safe-signs(patient)
```

becomes true. If two patients become unsafe at the same time, there will be two demon instantiations simultaneously.

Demons may take time to execute. What happens if the triggering predicate of the demon becomes true while the demon is still active? The answer depends upon whether or not the qualifiers **serial** or **parallel** are included. If **parallel**, a new demon instantiation is created, which executes in parallel with the previous one. If **serial**, the demon cannot be re-invoked on the same parameters until the previous instantiation of the demon is terminated (invocations on different parameters are allowed, however). If neither **serial** nor **parallel** is indicated, parallel is assumed.

I.5. Function Declarations

Function declarations are a convenient way of defining expressions which are used repeatedly in a specification. Functions take the form

```
function <name> ( { <parameters> } )
  { returns <type> }
  definition <expression>
```

The expression in the body of the function is assumed to be applicative. Functions are always evaluated instantaneously; there can be no state transition within a function execution.

Note that functions bear a similarity to implicit relations. For example, the following function definition and relation definition both define factorials.

```
function factorial(n|natural-number)
  definition
    if n = 0
      then 1
      else n * (factorial(n - 1))

implicit relation factorial(n1|natural-number, n2|natural-number)
  iff ((n1 = 0) and (n2 = 1))
      or (factorial(n1, n * (factorial(n - 1))))
```

I.6. Invariants

Invariants take the following form:

```
invariant <predicate>
```

The predicate in an invariant must be true in every state of every behavior.

The interpretation of an invariant depends upon whether the types and relations in the predicate are dynamic or static, and whether they are implicit or explicit. If any of the types and relations are dynamic and explicit, then the specification must define activities that make the predicate true; otherwise the behavior is disallowed. If all of the types and relations are static or implicit, then stating the invariant is sufficient to guarantee that the invariant holds, assuming that it is not inconsistent with the rest of the specification. No additional behavior definition is required.

An example will make the distinction clear. Suppose that we have the following invariant appears in a specification:

```
invariant for all n|nurse ||
  exists w|ward || assigned-to(n, w)
```

This invariant states that for every nurse there exists a ward that that nurse is assigned to. Suppose that **nurse**, **ward**, and **assigned-to** are all dynamic and explicit. This means that nurse and ward objects exist in a behavior only if some activity creates them; **assigned-to** is true only if explicitly asserted. Any behaviors in which nurses are created but no corresponding **assigned-to** assertion exists are treated as anomalous; they are excluded from the set of valid behaviors. Now suppose that the **nurse**, **ward**, and **assigned-to** are all static. Then no behavior can create the types, or assert the relation. Instead, a different initial behavior state is presumed to exist for each configuration of nurses, wards, and ward assignments that satisfies the invariants.

If dynamic and static relations are mixed in an invariant, the invariant may or may not be statically guaranteeable; it depends upon the invariant. In order to keep the interpretation clear, we try to avoid mixing dynamic and static relations in invariants.

1.7. Procedures

Procedures are what used to be called "actions" in earlier versions of Gist. They have the following form:

```
procedure <name> [ { <parameters> } ]
    returns <types> }
```

`definition` *<statement>*

Procedures are used to define activities and events. They are executed only when explicitly invoked. Procedure definitions consists of primitive database operations that create or destroy objects, or which delete, insert, or update relations. These contrast with functions, which cannot modify relations or objects. Procedures can optionally return one or more values.

References

1. Balzer, R., D. Cohen, W. Swartout. Tools for Specification Validation and Understanding. Tech. Rept. RADC-TR-292, Rome Air Development Center, December, 1983.

2. Balzer, R. M. Living in the Next generation Operating System. Proceedings of the 10th World Computer Congress '86, IFIP, Dublin, Ireland, September, 1986. (Accepted for publication).

3. Barstow, D.. *Knowledge-Based Program Construction*. North Holland, 1979.

4. Cheatham, T. "Reusability through program transformations". *IEEE Trans. on Software Engineering SE-10*, 5 (1984), 589-595.

5. Cohen, D. Symbolic execution of the Gist specification language. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, IJCAI, 1983, pp. 17-20.

6. Cohen, D. A Forward Inference Engine to Aid in Understanding Specifications. Proceedings of the National Conference on Artificial Intelligence, AAAI, August, 1984, pp. 56-60. Also available as USC-ISI/RS-84-135.

7. Cohen, D. *AP5 Manual*. USC-Information Sciences Institute, 1985. Draft.

8. Darlington, J. "An experimental program transformation and synthesis system". *Artificial Intelligence 16* (1981), 1-46.

9. DeMarco, T.. *Structured Analysis and System Specification*. Yourdon Press, 1978.

10. Feather, M.S. Constructing Specifications by Combining Parallel Elaborations. To appear in IEEE TSE.

11. Goldman, N. and D. Wile. Gist Language Description. Draft.

12. Goldman, N. *AP3 Reference Manual*. USC Information Sciences Institute, 1983.

13. Green, C., D. Luckham, R. Balzer, T. Cheatham, C. Rich. Report on a Knowledge-Based Software Assistant. Tech. Rept. RADC-TR-83-195, Rome Air Development Center, August, 1983.

14. Johnson, W.L. Turning ideas into specifications. Proceedings of the Second Knowledge-Based Software Assistant Conference, Rome Air Development Center, 1987.

15. Manna, Z., and Waldinger, R. "Synthesis: Dreams => Programs". *IEEE Trans. on Software Engineering SE-5*, 4 (July 1979).

16. Neches, R., W. Swartout, J. Moore. "Enhanced Maintenance and Explanation of Expert Systems through Explicit Models of Their Development". *Transactions On Software Engineering 10* (November 1985). Revised version of article in Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems, December, 1984.

17. Sanders Associates. Knowledge-Based Requirements Assistant - Interim Technical Report. Software Systems Engineering Directorate, March, 1986.

18. Swartout, W. GIST English Generator. Proceedings of the National Conference on Artificial Intelligence , AAAI, 1982.

19. Wile, D. Program Developments: Formal Explanations of Implementations. Communications of the ACM, 1983, pp. 902-911. (Also published USC/Information Sciences Institute RR-82-99, 1982).

Performance Estimation for a Knowledge-Based Software Assistant*

Allen Goldberg
Kestrel Institute

Douglas R. Smith
Kestrel Institute†

June 23, 1987

1 Goals of a Performance Estimator Assistant

As defined in [2], the long-term goal of a Performance Estimator Assistant (PEA) is to aid in the creation and maintenance of programs that meet their *performance requirements* or, if specific performance requirements have not been stated, are generally efficient in their use of computational resources. The principal function of the Performance Estimation Assistant is to guide implementation decisions that will affect the ultimate performance of the system, by generating and reasoning about performance estimates for expressions ranging from very-high-level specifications to low-level code. For example, efficiency estimates can be used to choose between alternative low-level implementations of high-level, set-theoretic data structures [4]. Efficiency estimates also provide the basis for choosing among alternative control structures, optimization transforms, library routines, and so on. Performance estimation over a rich space of alternative implementations is the feature that differentiates a practical program synthesizer from an executable specification language good only for prototyping.

The short-term goals for PEA capabilities called out in [2] are:

- symbolic evaluation [3] [1],
- data structure analysis and advice [4], and
- subroutine and module decomposition advice.

*This research was supported by the Rome Air Development Center (RADC) under contract F30602-86-C-0026. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied of RADC or the U.S. Government.

†1801 Page Mill Road, Palo Alto, CA 94304.

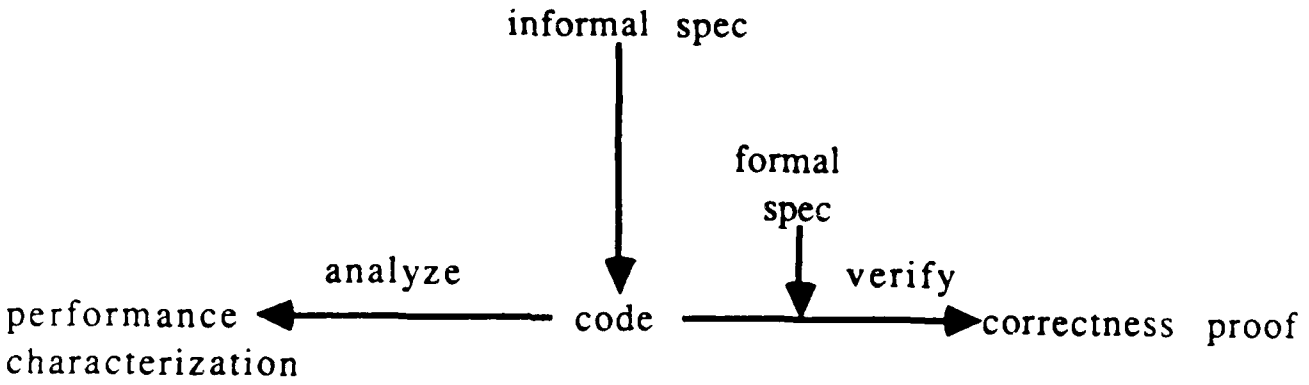


Figure 1: Current Paradigm

Mid-term goals are:

- domain models for analysis,
- algorithm design analysis and advice, and
- real-time performance advice.

2 Performance Analysis and Paradigms for Software Engineering

In the conventional paradigm of software engineering, depicted in Figure 1, code is developed in an intuitive and largely informal manner from requirements and specifications. The resulting code is typically incorrect and has unknown performance characteristics. To ascertain the correctness of the code we test it with examples or verify it against a formal specification. The latter has the advantage of mathematical rigor and greater certainty in the correctness of the code, although all that we have done is show consistency between the specification (which may be incorrect) and the resulting code, but it is prohibitively expensive to do this in most cases. Likewise if we are interested in the performance of the code then we usually simply test it and measure its performance. Sometimes we perform analysis of its properties and/or properties to get an assessment of resource utilization. Again it is expensive to do this in many cases.

The transformational paradigm (see Figure 2) finesses these issues by developing code and properties incrementally from specifications by a sequence of correctness-preserving and analysis-updating transformations. The resulting code and analysis is guaranteed to be correct modulo the initial specification (and modulo the correctness of the transformations

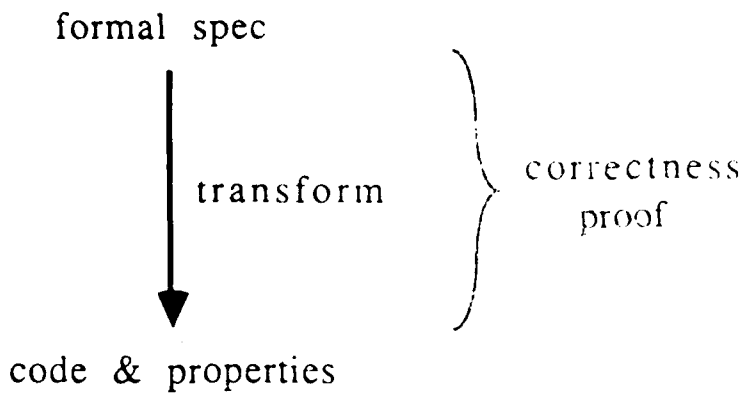


Figure 2: Transformational Paradigm

used). From the point of view of performance analysis, another advantage to the transformational approach is that performance information is available to help determine which transformations to apply (and thus provide search guidance).

3 Role of the Performance Estimation Assistant in the KBSA

The economics of computing forces performance to play a central role in the software lifecycle. We believe that the PEA will provide valuable assistance to most, if not all, facets of a KBSA. For example, the PEA can be used by the Requirements Assistant to help assess the feasibility of the desired behavior with respect to the available computational and organizational resources. Performance can be used by the Specification Assistant to help assess alternative modularizations of the target system. Performance can be used by a Test-and-Integrate Assistant to help validate timing goals and constraints between modules and overall system performance.

The most immediate and essential use of the PEA will be by the Development Assistant. While there are many activities that one would eventually want supported by a PEA, the one we are focussing most of our attention on is search guidance during development. The development of code from specifications can be simply modelled by a tree structure where nodes represent partially implemented specifications and arcs represent transformations (See Figure 3). Paths through the tree represent derivations where the final nodes represent executable target-language programs. Generally there are alternative transformations (arcs) that are applicable to any given specification (node) each potentially leading to different implementations of the initial specification. Depending on the granularity of our transformations, there will typically be many transformations involved in a derivation so the tree of

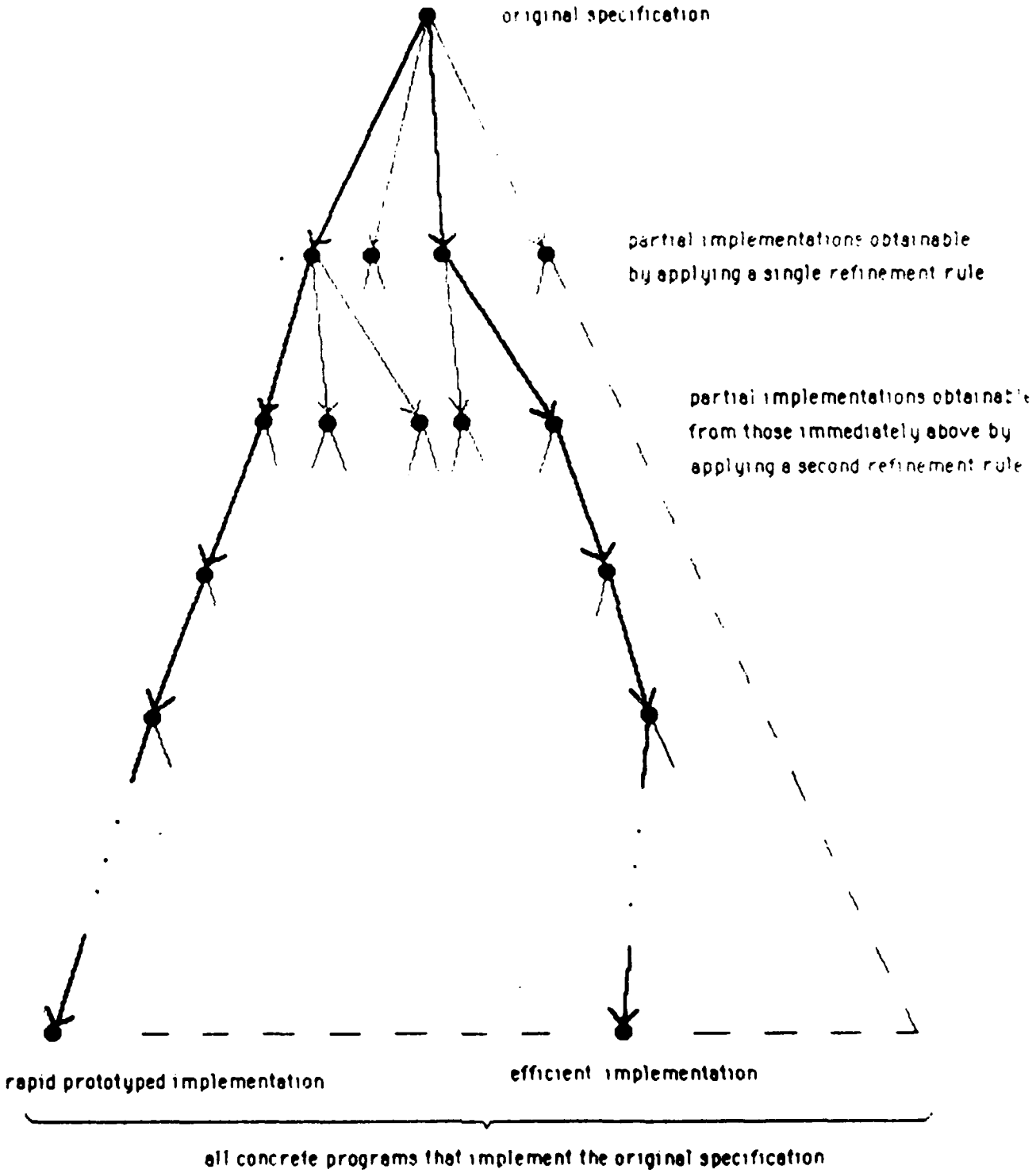


Figure 3: Development Tree

alternatives derivations can grow quite large, too large in fact to exhaustively [illegible]

A crucial role of the PEA is help guide the search through this tree. The alternatives that can be derived are distinguished mainly on the the basis of their resource consumption, they all implement the same functionality (since transformations are correctness preserving). Thus a PEA serves to help decide which transformations are more likely than others to lead to an acceptably efficient program.

4 Requirements for a Performance Estimation Assistant

A system whose use costs more than the benefit that it produces is clearly an uneconomic enterprise. Concern for the resource consumption of a software system falls under the general rubric "performance estimation and analysis." The PEA can be viewed as a collection of tools that help to produce systems that efficiently use available resources.

Uses of a PEA include:

1. Choosing between alternatives during development. This was discussed above.

2. Identifying bottlenecks. The development facet needs to identify portions of a specification that concentrate resource utilization (i.e. bottlenecks), since they are often a profitable target of optimization transformations. By helping to identify such bottlenecks, the PEA helps with the efficient scheduling and allocation of resources at development time.

3. Characterizing the resource utilization of a system. We may want an explicit characterization of the resource utilization of a given specification, for documentation purposes, performance prediction, explanation, etc.

4. Determining if a system meets performance constraints. Part of the initial specification may be constraints on the resource utilization of the target code. The PEA should understand these constraints and be able to reason about them and propagate their consequences to various parts of the system. The PEA should help to ensure that such constraints are met.

5. Analyzing a system for various properties and relationships. Underlying the ability to measure resource utilization and to guide decision-making during development is the ability to detect and infer properties of the system and relationships between its parts.

6. Tracking and controlling the resource consumption of the development process. The development process itself consumes resources and there is a need to monitor these resources, and, as part of decision-making, help determine an effective allocation of resources among various development tasks and to help schedule them. There is a tradeoff between the resources consumed during design and the resource consumption

of the target system. Generally it takes longer to find and develop a more efficient program.

5 Approaches to Performance Analysis

We can distinguish four approaches of performance analysis; namely, qualitative (heuristic), quantitative (statistical), simulation, and symbolic. We discuss the various characteristics of these approaches below and how they might be implemented.

There are several performance-related dimensions that we can use to help characterize these approaches.

- What resources are of interest? Often we want to minimize consumption of time, space, size of particular data structures, processors, communication, or some combination of these.
- How is the initial environment characterized? That is, if we are interested in the time complexity of a program, then we need to measure time as a function of some aspect of the initial environment, particularly the input; e.g. number of bits to encode the input, set size, probability measure, characteristics of a typical input.
- What statistic is of interest? Are we interested in best case, worst case, average case, typical case analysis, amortized worst-case, or some other?
- How precise should the analysis be? It may be that a coarse feature space is sufficient for decision-making purposes. In other cases we may need asymptotic, or even more precise, microanalysis of the target code.
- How accurate should the analysis be? It may be that an estimate is sufficient for decision-making purposes. In other cases we may need exact characterizations of resource utilization.
- How general should the analysis be? Some kinds of analysis only characterize the resource utilization of a restricted subset of the problem domain, while others yield more general results.
- How expensive is the performance analysis task? There is a tradeoff between the time it takes to assess performance and the precision, accuracy, and generality of the results. If we need to develop a program quickly, then a fast but imprecise approach may be best.
- What program properties does performance depend on? Analysis of program properties seems to be a crucial capability underlying all of the approaches to performance analysis/estimation and decision-making during development.

type of analysis	generality	precision/ accuracy	[illegible]
qualitative (heuristic)	general	imprecise	[illegible]
statistical (quantitative)	special	range of accuracy	medium
simulation	special	precise	medium to high
symbolic	general	range of precision & accuracy	high

Table 1: Approaches to Performance Analysis

What kinds of information are needed to support the above dimensions of performance estimation? There are at least two kinds. One is problem-independent knowledge, e.g. about the performance of various "primitive" operators in the specification language. The other is problem-specific and must be supplied in the requirements and specification. Problem-specific information includes any constraints on the desired performance of the target system and any assumptions that can be made about the environment. Environmental assumptions could include a probabilistic measure of the input domain or the size of a typical input or the relative frequency with which various operations are expected to be performed.

We now describe four general approaches in terms of the above dimensions. Table 1 summarizes this discussion.

- *Qualitative* or *heuristic* performance analysis is based on qualities of a specification rather than quantities. For example, data structure implementation decisions depend on the 'mix' of operations applied to the data structure. This is a *qualitative*, not quantitative property of the program. It may be unnecessary to determine the exact

size of the data structure, so simple heuristic reasoning would often suffice to guide decision-making. Qualitative reasoning is particularly appropriate when good performance decisions can be made on the basis of relatively coarse analysis of the program, for example, when the presence or absence of certain features provides the information needed to choose generally efficient implementations. The cost of qualitative analysis tends to be low since it does not rely on general-purpose inference or deep analysis. By its nature the results of qualitative analysis are imprecise and often inaccurate—but in certain domains that is enough to lead quickly to good implementations most of the time.

- *Quantitative* or *statistical* performance analysis is based on statistical properties of the *environment*, the primitive operators, estimates of such quantities as data structure sizes and selectivity of predicates. Quantitative analysis often involves typical case analysis which has limited generality, but may be indicative of general performance. The precision and accuracy of its results vary depending on the precision and accuracy of the underlying statistics and methods for inferring the statistics of aggregate structures. Typically, quantitative analysis runs in time linear in program size, so it is relatively inexpensive to perform

- *Simulation-based* performance analysis is based on statistical properties gathered by actually running the specification on test data. Since we only get information about the specifications behavior on a few inputs, this kind of analysis has low generality, but very precise and accurate measurements are possible. The cost of simulating can be relatively high, depending on how long it takes to instrument and prototype the specification, and to actually run it.

- *Symbolic* performance analysis is potentially the most powerful approach. It is based on deep analysis of program properties and results in general symbolic characterizations of resource utilization. Varying degrees of precision are possible, but at a cost. The drawback to symbolic analysis is that in general it is unsolvable, and for many restricted domains it is computationally intractable.

6 Toward Realizing a PEA

We analyze the general capabilities required of a PEA. These capabilities include a knowledge-base representation of performance-related information, demand and data-driven analysis modes, and the ability to instrument a specification for metering. As described earlier, specifications will need to include performance constraints (or goals) and any assumptions that can be made about the run-time environment of the target system. The knowledge-base serves to represent the functional constraints (which define the desired functionality of the target system) along with annotations that describe the performance constraints and environmental assumptions. These annotations may be automatically propagated down into the abstract syntax of the functional constraints by *constraint-propagation demons*. Only simple, relatively cheap, but generally useful inferences would be made via this *data-driven* mode. Other kinds of analysis are more expensive to make and/or more rarely useful and so

there must be some explicit outside stimulus to perform the
from the initial specification, from the user or from a
it can be applied. The idea is that various analyses
sense. The knowledge-base allows queries from users
interested facet regarding performance knowledge of the

When a specification is transformed, the result is a
Performance analysis typically needs to be done in order
tion invariants. The representation is then reanalyzed in order
annotations up to date—to support further decision making
able to respond to the application of a transformation
annotation invariants have been conventionalized.

In summary, we believe that a general PEA will support
the knowledge-base manager

- demons which automatically propagate
- demand-driven analysis and propagation
- maintenance and querying of performance
- automatic instrumentation of a specification for

7 Example: Performance Analysis during Algorithm Design

We have explored a design rules for transforming specifications into divide-and-conquer algorithms [7] conditional programs [6], generate and test algorithms [5] and others. We illustrate our approach to performance estimation and several guidelines by an example. First we briefly describe a design rule for divide-and-conquer algorithms then show how it can be extended with performance estimation/analysis capabilities and finally describe the derivation of a mergesort algorithm.

A design rule for divide-and conquer algorithms derives an instance of the following program functional scheme:

$$DC(x) ::= \textbf{if } Primitive(x) \\ \textbf{then } Directly-Solve(x) \\ \textbf{else } Compose \circ [G \times F] \circ Decompose(x)$$

where G may be an arbitrary function, but typically is either the identity function Id or F. $f \circ g$, called the composition of f and g, denotes the function resulting from applying f to the result of applying g to its argument. $[f \times g]$, called the product of f and g, is defined by $[f \times g](x, y) = < f(x), g(y) >$. *Decompose* is referred to as the decomposition operator, G is referred to as the auxiliary operator, *Compose* is referred to as the composition operator, and *Directly-Solve* is referred to as the primitive operator. In order to construct an instance of this scheme: First choose a simple decomposition operator, and choose an auxiliary operator, then use constraints to derive input and output conditions for the composition operator.

The performance of a divide-and-conquer algorithm can also be described by means of a scheme. Let $S_F(n)$ denote the maximum size of an output of program F as a function of inputs of size n. Similarly, let $T_F(n)$ denote the maximum amount of time required by program F to compute an output for an input of size n. The worst-case output size and time complexity of a divide-and-conquer algorithm are described by the schematic recurrence relations:

$$S_{DC}(n) = \begin{cases} S_{Directly\text{-}Solve}(n) & \text{if } Primitive \\ S_{Compose} \circ [S_G \times S_F] \circ S_{Decompose}(n) & \text{if } \neg Primitive \end{cases}$$

$$T_{DC} = \begin{cases} T_{Directly\text{-}Solve}(n) & \text{if } Primitive \\ T_{Decompose} + Plus \circ [T_G \times T_F] \circ S_{Decompose}(n) + \\ \quad T_{Compose} \circ [S_G \times S_F] \circ S_{Decompose}(n) & \text{if } \neg Primitive \end{cases}$$

These schemes are used in two ways. First, for analysis purposes: after a divide-and-conquer algorithm has been constructed, the schemes are instantiated and the recurrences solved to give an exact output-size and time analysis. Second, for estimation purposes: if we have a performance bound to be met and a simple decomposition operator has been chosen (according to the design rule) then the recurrences can be used to determine performance bounds on the specification for the composition operator.

Now we may apply this to a particular problem. Suppose that we start with the following specification for the problem of sorting a sequence of numbers:

$$SORT(x) = z \text{ such that } Permutation(x, z) \wedge Ordered(z).$$

Suppose furthermore that we want to derive a sort algorithm whose worst-case asymptotic time behavior is optimal. Suppose furthermore that a $O(n^2)$ sort algorithm has been derived elsewhere in the search tree. Now, following the design rule we begin to design a divide-and-conquer sort algorithm by choosing a simple decomposition operator on sequences. Suppose that the decomposition operators FirstRest and ListSplit are known to the system and have the following performance information associated with them:

$$S_{FirstRest}(n) = (1, n-1) \quad T_{FirstRest}(n) = 1$$

$$S_{ListSplit}(n) = (n/2, n/2) \quad T_{ListSplit}(n) = n$$

Suppose that we choose FirstRest. This choice leads to the desired [illegible] algorithm. The design rule performs some formal manipulation [illegible] specification for the composition operator (which inserts a number into a sorted list). In parallel the extended rule would derive a bound on the performance of the program as follows. Instantiating the time analysis scheme we obtain

$$T_{SORT}(n) = 1 + 1 + T_{SORT}(n-1) + T_{Insert}(1, n-1)$$

where T_{Insert} is the time complexity of the composition operator (here called Insert). T_{SORT} must better $O(n^2)$ we plug n^2 in for $T_{SORT}(n)$ and solve for an upper bound on T_{Insert}

$$n^2 = 1 + 1 + (n-1)^2 + T_{Insert}(1, n-1)$$

or

$$T_{Insert}(1, n-1) = n^2 - (n-1)^2 + 2 = 2n$$

So in order to produce a sort algorithm better than $O(n^2)$, an Insert algorithm better than $O(n)$ must be synthesized. A short way into the derivation of Insert we note that this is impossible (Simple algorithms for Insert take at least $O(n)$ time). So this line of development can be terminated and attention focused elsewhere.

Suppose that we backup to the design rule and choose ListSplit as decomposition operator. The analogous performance calculation for the composition operator (called Merge) yields a bound of $O(n^2)$ in order to obtain a sort algorithm faster than $O(n^2)$. This bound is easily achieved and in fact a $O(n)$ Merge algorithm can be derived using the same divide-and-conquer design rule [7].

An exact analysis for the completed mergesort algorithm is performed as follows. From the derivation of Merge we get

$$S_{Merge}(n, m) = n + m \qquad T_{Merge}(n, m) = n + m$$

Plugging these and other values into the analysis schemes for divide-and-conquer we have

$$\begin{aligned} S_{Msort}(n) &= S_{Merge} \circ [S_{Msort} \times S_{Msort}] \circ S_{ListSplit}(n) \\ &= S_{Merge} \circ [S_{Msort} \times S_{Msort}](n/2, n/2) \\ &= S_{Msort}(n/2) + S_{Msort}(n/2) \end{aligned}$$

which has solution $S_{Msort}(n) = n$.

$$\begin{aligned} T_{Msort}(n) &= T_{ListSplit}(n) + Plus \circ [T_{Msort} \times T_{Msort}] \circ S_{ListSplit}(n) + \\ &\quad T_{Merge} \circ [S_{Msort} \times S_{Msort}] \circ S_{ListSplit}(n) \\ &= 1 + 2T_{Msort}(n/2) + T_{Merge}(n/2, n/2) \\ &= 1 + 2T_{Msort}(n/2) + n \end{aligned}$$

which has solution $T_{Sort}(n) = O(n \lg n)$. In summary, this example shows

1. how performance estimation capability can be built into a design rule;
2. how these estimates can be used to guide and prune the search space;
3. how to perform exact performance analysis on compiled programs.

8 Future Work

Clearly there is much work that remains to be done before a general robust PEA emerges from the laboratory. At the end of the current project we expect the following results.

1. Demonstration of the feasibility of the qualitative, statistical, and symbolic approaches to performance estimation;
2. Demonstration of the feasibility of fully automatic tools for performance estimation;
3. Deliverable prototypes for guiding the development of low-level optimizations, data structure selection, and control structure optimization;

References

[1] Thomas E. Cheatham, Jr., Judy A. Townley, and Glenn H. Holloway. A system for program refinement. In *4th International Conference on Software Engineering*, pages 53-62, IEEE, Munich, West Germany, September 17-19, 1979.

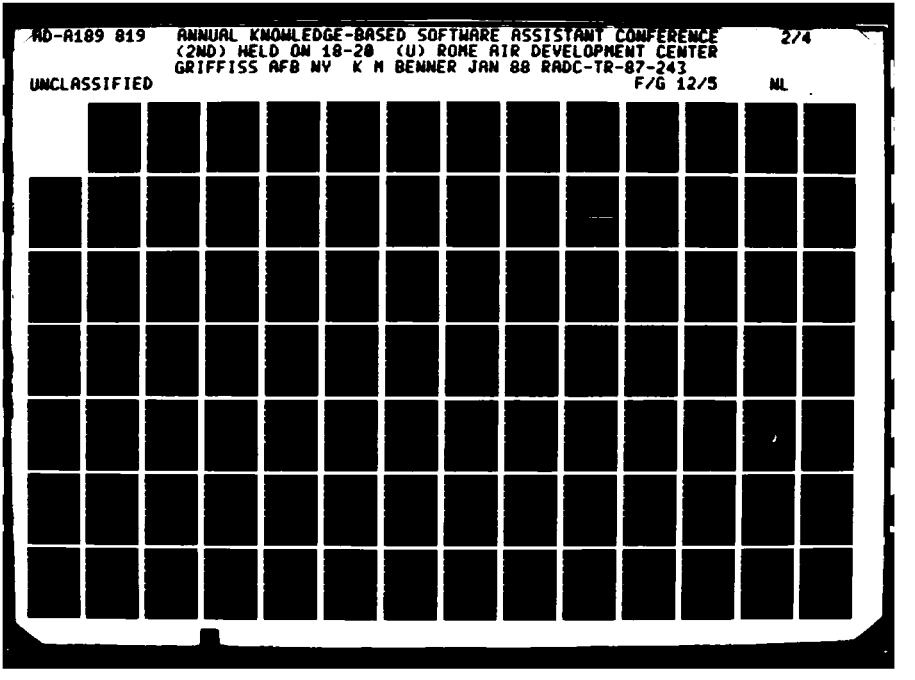
[2] Cordell Green, David Luckham, Robert Balzer, Thomas Cheatham, and Charles ... *Report on a Knowledge-Based Software Assistant*. Technical Report KES.U.83.2, Kestrel Institute, July 1983.

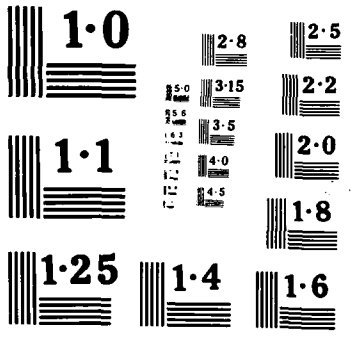
[3] Elaine Kant. *Efficiency in Program Synthesis*. UMI Research Press, Ann ... 1981.

[4] Gordon Kotik. *Knowledge-based Compilation of High ...* port KES.U.83.5, Kestrel Institute, March 1983.

[5] Douglas R. Smith. On the design of generate... In L. Meertens, editor, *IFIP TC 2 Working ... Transformation*. North-Holland, Amsterdam, ...

[6] Douglas R. Smith. Reasoning by ... *Proceedings of the Ninth International ...* Angeles, CA, August ... 1986. Technical ... Kestrel ...


AD-A189 819 ANNUAL KNOWLEDGE-BASED SOFTWARE ASSISTANT CONFERENCE (2ND) HELD ON 18-20 (U) ROME AIR DEVELOPMENT CENTER GRIFFISS AFB NY K M BENNER JAN 88 RADC-TR-87-243
2/4
UNCLASSIFIED
F/G 12/5
NL


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0
5·6

[7] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, February 1985. Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.

KBSA Perspective Panel
Introduction

I. How has KBSA Vision Changed?
  A. Vision 5 years old
  B. Vision as stated in KBSA report:
    1. New Paradigm
      a. Informal person-based => formalized computer-assisted
      b. "Machine-in-the-loop"
      c. Build "Corporate Memory" of decisions rationale
        Use for development maintenance
      d. Requires knowledge-based assistant
    2. New Software Life Cycle
      a. Formal Specification
        Executable as prototype
        Validated via usage
      b. Implementation
        Derived from formal spec via transformations
        Testing disappears
      c. Maintenance
        Modify formal spec rederive implementation
    3. Major Effects
      a. Systems will be larger, more integrated, longer lived arise from many small evolution steps
      b. Software will finally remain "soft"
      c. Evolution will become central software activity
      d. Evolution will also become basis for initial development
  C. Vision today
    1. Remarkably intact (i.e. unchanged)
      For saw emergence of formal process model
    2. Some Laboratory Experience with parts of vision
      a. Executable Specifications
        Refine: Kestrel
        Setl: NYU
        AP5: ISI
        EL1: Harvard
      b. Corporate Memory
        FSD/CLF: ISI (maintenance context)
        EES: ISI (explanation)
      c. Evolved Specification (Prototype) Delivered
        Refine: Kestrel
        Ada Compiler: NYU
        FSD/CLF: ISI
      d. Knowledge-Based Development
        Programmer's Apprentice: MIT
    3. No experience with vision as a whole

4. New Insights
    a. Acquiring decisions rationale is hard
        Both representation elicitation
    b. Modifying Specifications is hard
        Lack of optimization only
            helps localize information
        Specs have structure
    c. Compiler Advice (annotations) is an alternative
        to transformational derivation

II. How is KBSA Program Progressing?
A. Program 3 years old
B. Program generally following plan
    1. Several independent efforts
        Each focused on a facet
        Each embodying a framework
    2. 4 of 7 identified facets funded
        Plus one about to be
        Most short term (3-5 year) milestones met
            Exception: subroutine module
                decomposition advice within
                Performance Facet
    3. Framework definition concept demonstration effort started
    4. Supporting technologies not explicitly funded
    5. Bilateral facet communication (batch) just being initiated
    6. Common problem domain selected (Air Traffic Control)
C. Prognosis
    1. Optimistic
        a. On track
        b. "Phase Coupled" integration feasible
        c. Overlapping development provides basis
            for comparison understanding
    2. Pessimistic (Hard issues yet to come)
        a. Integration
            Standards
                Framework: no closer to consensus
                    than before
                Common User Interface: not separated
                    from facet
                Mechanism not yet identified
            Interleaved operation
            Ability to process partial descriptions
        b. Engineering
            Distributed Framework
            Scaling

KBSA TRANSITION

Samuel A. DiNitto, Jr.

Historically, the transition of software engineering technology has lagged not only hardware engineering technology, but has even lagged advancements in software applications. We have examples of software engineering tools and techniques that demonstrated their worth in the early 1970's and are still not widely used, if at all. We have programming languages that began to be heavily used only about the time they became politically, if not technologically obsolete. We also find that, as new "software engineers" or software engineering components of established industries crop up, they make the same mistakes and often limit themselves to the same technologies used in the early 1960's, namely, a compiler (maybe) and a linking-loader (usually).

A cursory analysis of the reasons for this lack of techology transition leads to at least the following, overlapping reasons:

1. Unawareness of the technology.
2. "Culture shock." (That's not the way we did things in the past.)
3. Initial cost of tools training, etc.
4. Lack of "support" for the technology (maintenance, training, etc.)
5. Lack of hard data on cost vs. productivity improvement.
6. No push from the competition.
7. No "serious" demand from the customer for high quality or high productivity.
8. Incompatibility with the development hardware.

All of these reasons, and more, will be used against the KBSA technology.

Unless there is a very unlikely drastic change (for the better) in terms of a willingness to spend huge sums of money on large, "real" systems to try out, and prove, new technology, the transition of the technology will depend very heavily on the ability to make the KBSA paradigm widely available, at low cost. By "low cost" is meant the cost of the necessary hardware, software licenses, training, support, and what might be perceived as "additional" manpower needed to support the paradigm. Thus, this factor, which is a combination of 3, 4, and 8 above and which will be labeled "cost of use," is the key to transition or nontransition of this technology. With a low cost of use, all the other deterrents to software engineering technology transition can be overcome with additional effort; without it, the odds of successful transition are small.

If we look at the past, at a simple tool like the misnamed "path-tester" that provided a measure of how well the software was tested, the above

point is brought home. Discussions with software developers revealed that, even though the cost of a license (usually around $5K) was low, there was reluctance to spend the money. Even after the money was spent, the tool was often not used because additional manpower would be spent in using it, and, after all, the customer didn't require a demonstration that each piece of code was tested.

Software quality metrics technology, which attempts to take quantitative measures of those quality attributes held so dear, (reliability, maintainability, efficiency, portability, etc.), provides another recent example. When told that, for a cost equal to about five per cent of the software development expenditures, a system program manager could have a capability to specify, predict along the way, and assess the final software quality before acceptance, two kinds of responses were given. From those who saw the problems of many military software acquisitions, came responses like "cheap at four times the price." From those tied up in their own single system development came comments like "too expensive" or "outrageous"!

Finally (although not all that comes to mind), there is still today being fought the battle over the use of a high order language versus assembly language in certain military applications. The same arguments that have existed for over 20 years are made again:

1. The cost of the compiler (development, retargetting, license).

2. The cost of training and/or getting experienced people.

3. The cost of additional hardware to take care of the additional memory that will be needed.

Of course there are other reasons given, as well.

1. Immaturity" of the language/compiler.

2. Lack of personal experience (confidence) with the compiler/ language.

3. The people to be used have already delivered a similar system using assembly language or a different language.

However, without "breaking the ice" with that first use, which always equates to overcoming actual or perceived costs, one can never attack the latter set of obviously overlapping and related reasons.

The position taken is thus very simple. Without keeping the "cost of use" down to a reasonable level, the chance may never come to actually demonstrate the benefits of the KBSA technology. Or in other words, the chances for a successful technology transfer are inversely proportional to the perceived cost of use.

PROBLEMS AND PROSPECTS FOR TECHNOLOGY TRANSFER IN EXPERT SYSTEMS

ROBERT J. GLUSHKO

SOFTWARE ENGINEERING INSTITUTE

Technology can be defined as the application of knowledge to solve a practical problem. Thus, the development and deployment of knowledge-based or expert systems is a pure form of technology transfer in which the knowledge being transferred is made explicit.

Technology transfer is usually more difficult and often takes longer than either the developers of the technology or its recipients want to believe. Unfortunately, the challenges of technology transfer are even greater for expert systems, which face barriers above and beyond those that impede the successful development and deployment of conventional software systems. In this paper I discuss some of these special problems for technology transfer in expert systems, noting where possible the prospects for overcoming them.

The successful transfer of technology as knowledge embedded in an operational software system is the culmination of a long series of activities. Primarily, these include defining the "right" system requirements in response to the user's needs, developing the system that meets those requirements, deploying the system in the user's work environment, and then using and maintaining the system during its deployed lifetime. Each of these activities is more complex for expert systems than for conventional software systems.

Getting the requirements right is difficult for expert systems because both developers and users typically have unrealistic expectations about how much knowledge can be embodied and transferred. Many people have noted the "overselling of AI," but unfortunately some overselling was and is still inevitable, both for companies in the AI business trying to sell systems and for people within potential customer firms trying to generate management support for AI efforts.

Expert systems are new technologies, and part of the maturation of any technology is learning how and why it works along with its limitations. Overselling by expert system enthusiasts and entrepreneurs helped to establish the boundaries for suitable problems. Choosing the right problem for an expert system is essential, but the level of awareness and experience in most potential customer firms is low. Expert system vendors often need to "cultivate" customers to make them able to assess what they might be buying. Raising management awareness is an essential activity for vendors as a form of risk management, so that customers won't expect more than can be delivered. It is especially important to target upper management for "AI awareness," because these are the people who must provide support and sponsorship to make an expert system effort successful, and they are extremely unlikely to have any knowledge or experience about the potential and limits of AI. Strong management support will be critical during the knowledge engineering phase, because domain experts are always valuable and expensive employees of the customer organization, and the pressure is always there to return them to their "real" jobs where their expertise is needed now.

Once the customer achieves sufficient awareness to [illegible] with requirements definition for an expert system, two additional problems arise. It is hard to know how much knowledge engineering is necessary, because sometimes there just aren't good rules for solving a problem or because the experts don't agree on the rules. In addition, there are no good ways to measure how much expertise ought to be captured; without techniques to decide how much more an expert solution is worth, how can a customer decide how much to invest in capturing expertise? How much knowledge engineering is cost effective? Tools for determining return on investment in expert system problem domains are nonexistent.

Assuming that the customer and vendor can come up with a preliminary set of requirements, still other barriers to successful technology transfer emerge. Foremost is the "culture clash" between the developers of the expert systems and the domain experts. The stereotype of the AI hacker is receding, but many a potential sale has been lost because software developers were hastily sent to the customer site to give demos or talk to users. Expert systems have only recently begun to penetrate the commercial data processing world even though many activities in typical computer centers are perfect expert applications, such as capacity planning, performance tuning, and scheduling. The culture clash between the AI and DP worlds remains the major impediment.

Another barrier to successful technology transfer for expert systems is the reliance on hardware and software that is out of the mainstream of computing. LISP machines and special expert system development environments are essential to support the rapid prototyping that characterizes expert systems, but represent huge barriers to successful deployment. Some AI companies have already failed because they were wedded too tightly to proprietary hardware or software. Those who survive the shakeout will no doubt follow the developing trend to convert expert systems to run on conventional 32-bit machines and to replace LISP with C.

The problems with novel hardware and software are compounded after the system is deployed. Few customer organizations have pre-existing expertise to support LISP machines or expert system environments. More fundamental a barrier to technology transfer is the fact that "maintenance" problems in deployed expert systems are far more likely to be problems of knowledge engineering than problems of software engineering. That is, user problems are usually "buggy rules" rather than "buggy code." This situation requires that the system vendor train the users to be knowledge engineers so they can maintain their own systems. AI companies are making concerted efforts to develop "user-friendly" expert system shells with templates for rules so that "knowledge management" can be as commonplace as database management for the customer organization.

I hope that my litany of the problems that challenge the successful transfer of technology in expert systems is not discouraging. I noted that some of these problems are being overcome, or are being transformed into opportunities by innovative vendors and developers. Finally, let's hope that we all learn from each other at this conference.

I gratefully acknowledge the help provided by Dr. Laura Silver of the Carnegie Group and William Hefley of the Software Engineering Institute.

TECHNOLOGY TRANSFER

AND

THE KNOWLEDGE-BASED SOFTWARE ASSISTANT

Technology Transfer Panel
of the
Knowledge-Based Software Assistant Conference

Rome, New York
August 19, 1987

R. W. Lawler
Boeing Computer Services Co.

Technology Transfer and the Knowledge Based Software Assistant (KBSA)

Background

In 1983, Boeing established an Artificial Intelligence Center to take a leadership role in bringing AI technology to The Boeing Company. The center was to develop advanced technology and accelerate the infusion of that technology into the Boeing Operating Companies.

Boeing is a decentralized company with a small corporate headquarters and six operating companies. Boeing Commercial Airplane Company (BCAC), Boeing Aerospace Company (BAC), Boeing Vertol Company (BVC), Boeing Military Airplane Company (BMAC), Boeing Electronics Company (BECo), and Boeing Computer Services Company (BCS). The AI center was placed in the BCS Advanced Technology Center for Computing Sciences (ATC). In 1984, a company study group was formed to determine how the Boeing operating companies could make best use of AI technology. This study produced two major results: an identification of six initiatives that would exert a major pull on AI technology and a concept for technology transfer management.

One of these initiatives was the Knowledge-Based Software Engineering Initiative which is similar to the RADC KBSA. This initiative produced the "Crystal" technology which was demonstrated at the KBSA conference in 1986.

Figure 1 illustrates the model for management of technology transfer. Figure 2 represents an instance of this model in action for application of AI to maintenance and diagnosis. Note that in less than one year the technology was transferred to all operating companies; that technology was being shared between operating companies; and, feedback occurred that influenced the ATC designed based diagnostics R&D project. This bottom up rapid deployment of technology occurred because the theory and tools were sufficiently mature for this application area, and there were no barriers to the use of the technology. The KBSA does not enjoy a similar environment. Development and transfer of this technology will require a more managed approach.

Model of ATC's Technology Transfer Program

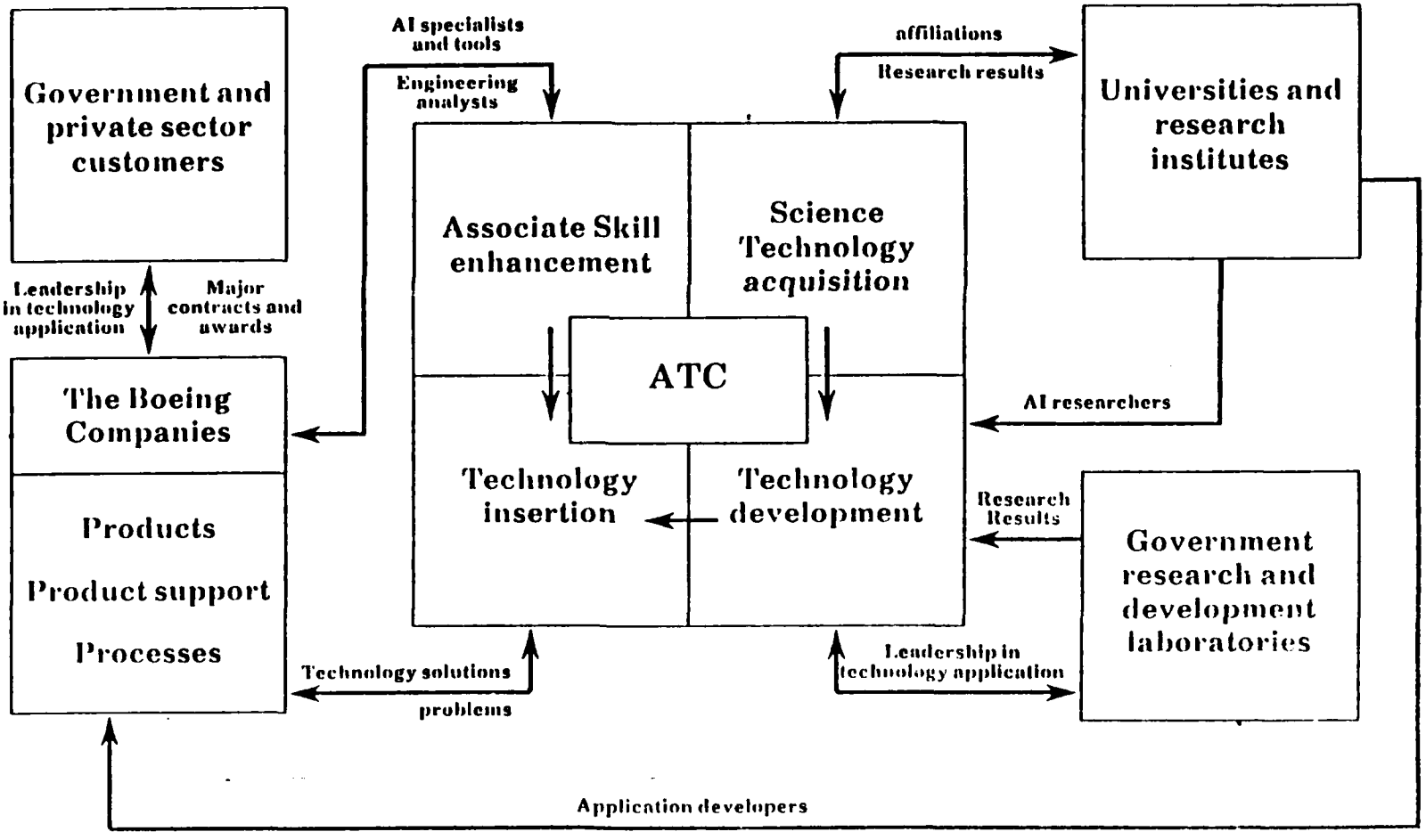


Figure 1

Technology Transfer at Work – Example 3

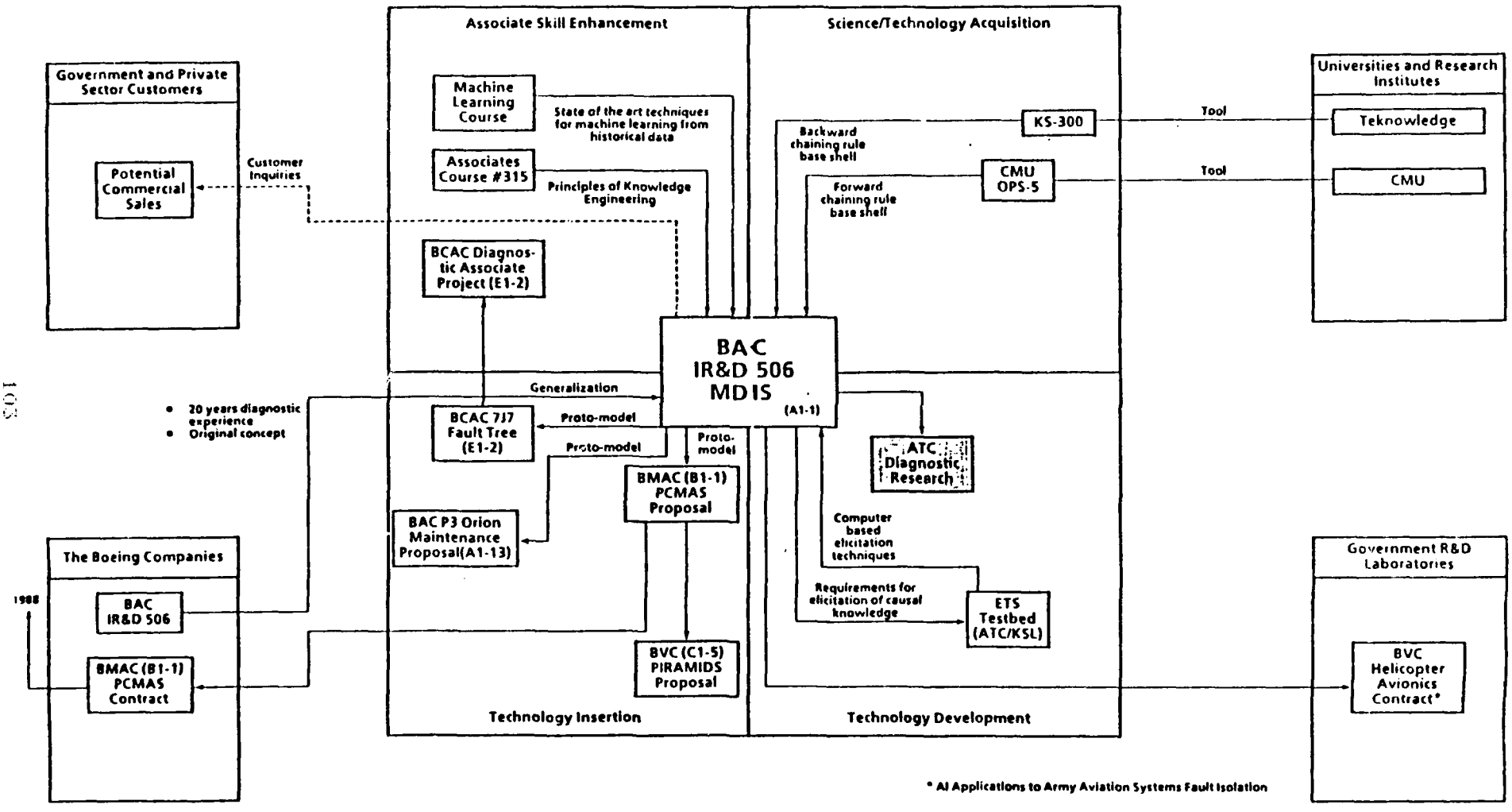


Figure 2

KBSA Technology Transfer Issues

The following issues will be discussed in the panel presentation.

Issue 1 - Technology Packaging

The concept for technology packaging affects the rapidity of transfer. Figure 3 illustrates this. The extremes are transfer to other R&D labs of research results and transfer of technology packaged in systems for building applications. Which parts of KBSA should be packaged at what level?

Issue 2 - Incremental vs Revolutionary Improvement in Software Productivity

KBSA technology has the potential to improve software development productivity significantly by radically changing the way we develop and maintain software. It can also be fielded as incremental improvements in current software development automation tool sets. A radical change will require major revisions to procurement regulations and standards. Will a radical change be more beneficial than incremental improvement?

Issue 3 - Commercial vs Military Applications of KBSA

Military procurements of new weapon systems require diverse leading edge technologies that impact one another. This impact is "discovered" during the procurement process and requires formulation of innovative concepts for applying technology to provide new functionality and performance. Is KBSA technology better proved in this high risk environment or in a commercial environment that requires few interacting technological innovations?

Issue 4 - Model for Transfer of KBSA Technology

The ATC model has been quite successful as a "managed" approach to technology transfer. What are some of the alternative ways this model could be modified to apply to transfer of DOD sponsored technology to industry and/or sharing of industrial development of the technology?

Technology Packages	Customer	Use	Conditions
Research Papers	Research Labs	Idea Stimulator	Mutual Interchange
Technology Demos	R&D Labs	Evaluation	o Protection of Proprietery Data o Engineering Compatibility
Functional Capabilities	Tool Developers	Incorporation into System	o Sales, Engineering Support o Reasonable Documentation
Systems	Software Engineers	Application Development	o Fully Supported. o High quality documentation o Customer Engineering

FIGURE 3 TECHNOLOGY PACKAGING ENVIRONMENT

SELECTION OF INVESTMENT STRATEGIES WHICH ARE OPTIMAL FOR THE IMPLEMENTATION OF SOFTWARE SUPPORT ENVIRONMENTS

ABSTRACT

Kenneth E. Nidiffer

Software Productivity Consortium

The objective of the research was to further the depth of understanding of the variables that impact the problem-solving process leading to the investment strategy decision. An investment strategy decision is made for almost every defense system acquired by the Department of Defense (DoD) with respect to the identification of what software support environment will be used in development and in support of the mission code associated with the defense system and what level of standardization will be employed by DoD with respect to the selected software support environment. Numerous kinds of software support environments have emerged in recent years to help solve problems being experienced by developers and supporters of software-intensive defense systems. Although modern software support environments increase productivity, they are expensive to build, deploy, and support and are often unique to the specific defense system under development. The investment strategy decision process is concerned with achieving the optimal balance between productivity and life-cycle costs. The government defense system program manager is required to form a working group to support the investment strategy decision process; however, this process in supported by many ill-structured activities that inhibit effective decision making. A need existed to perform research directed at answering two questions: (1) what is the proper investment strategy decision process and (2) what is the proper investment strategy decision? This research addressed these questions by (1)

developing an econometric model based on insights gained about the variables that impact the investment strategy decision process and (2) designing, building, verifying, evaluating and validating a program manager's decision support system. The decision support system, which is named the " *D*oD *S*oftware *I*nvestment *D*ecision (DSID)." model incorporates the econometric model and operates on a set of independent data files that can be updated as new information becomes available. The DSID model supports the program manager by providing a frame of reference to construct the problem and a method to perform "what if" exercises. There was no attempt to change the process only to provide the program manager with a decision aid that encompasses more formalism by which unstructured and/or semi-structured problem-solving activities are provided with structure. It is too early to judge whether the DSID model will prove a successful support system; however, considering the favorable results obtained from typical users and from evaluation of investment strategy decisions made on several defense programs, it is argued that the DSID model has significant potential with respect to improving the investment decision process. The DSID model's point of view is decidedly that of the government program manager and his superiors, but analysts and industry program managers who wish to understand some of the pressures that effect standardization, productivity, and life-cycle costs in the DoD community may also find this research of interest.

KBSA Common Framework Implementation

Aaron Larson and Steve Huseth
Honeywell Systems and Research Center
3660 Technology Drive
Minneapolis, Minnesota 55418

Abstract

This paper describes the implementation details of the KBSA Common Framework. The system forms a distributed knowledge base for knowledge-base programming tools and executes on multiple Symbolics workstations. Within the system, we have attempted to merge together an object-oriented programming environment supporting the representation of project data, with a logic programming environment to support data consistency and control of the programming process. We describe the internal organization of the system and some of the implementation decisions we have made. We conclude with future capabilities we plan to introduce.

1 System Overview

The KBSA Common Framework is written in Common Lisp, and currently runs on Symbolics workstations. The prototype is still in its early stages of development and therefore leaves many issues unaddressed. In writing this paper, we describe the strategies we have used to obtain a distributed framework and some of the design decisions we have made. A more complete discussion of the underlying motivations can be found in [Huseth87].

The prototype combines several programming paradigms into a single system by combining object-oriented programming via CommonLoops [Bobrow85], logic programming via LogLisp [Carciofini86], and imperative programming using Common Lisp [Steele84]. Each of these components have been combined in a dynamic distributed system. The framework supports object instance distribution, and a minimal capability to dynamically distribute object schema. Objects created on one workstation can be accessed and modified on any workstation in the network. All the other systems will automatically receive the updates. The collection of objects making up the system is referred to as the object base. Loglisp provides the ability to enforce logical relationships between objects, trigger demons satisfying logical predicates, and query for objects using the object attributes.

2 Object Distribution

Distribution of objects in the framework is done at the object level. User defined objects, called network objects, may be distributed and all of their slot values uniformly maintained across the network. A network object slot may contain value types of integers, strings, symbols, Lisp cons cells, or other network objects.

For simplicity, the object base is maintained on a central knowledge base called a server. The workstations that make use of the central server are called clients. To obtain efficient access to object instances, local copies may be requested by the clients. When a client first references an object, the object with all its slot values are transferred from the server to the workstation. When a slot value within the object contains another network object, only a skeleton of the referenced network object is created in the client. The skeleton object is identified as being incomplete and must be retrieved from the central server before any of its slots can be accessed. This prevents the whole object base from being sent when a highly connected object is

requested. Once an object has been referenced for the first time, subsequent accesses are made to the local copy of the object.

Modifying existing objects initiates actions which are required to maintain consistency between the local copies of the object instances. When an existing local object is modified, the change is made locally and a message is sent to the server to modify the central copy. The server then broadcasts a message to all the other workstations to invalidate their local copies of the object. The next time the invalidated object is referenced by any of the other clients, the new copy will be retrieved from the central object base.

Since network objects in a Lisp world may be manipulated like any other Lisp object (i.e. placed in lists, arrays, etc.) we must take care to ensure that the uniqueness properties of objects is maintained. Since we control the distribution and manipulation of network objects, we use a network object handle to uniquely identify an object. Primitive objects such as integer, string, symbols, and Lisp cons cells that are directly manipulated by Lisp cannot be guaranteed the same rigorous level of network transparency. For example, we do not maintain symbol values, properties, or function definitions across the network. Symbols stored in a network object slot must therefore only be used for their identity (i.e. **EQness**). Similarly we only maintain = for integers, **STRING-EQUAL** for strings, and **EQUAL** for conses. Any side effecting operations on strings or Lisp conses that are used as slot values will not necessarily be retained by the object base and as such are not legal ways to modify an object.

3 Integrating Objects and Logic

The programming environment supports the use of objects, methods and instances as well as the description of the objects using logical assertions. Each of these capabilities are discussed further.

In an object-oriented system, there are three basic entities: classes, methods, and instances. Each of these entities are supported by a the full object-oriented programming style provided by CommonLoops on a single workstation. Distribution of these components among multiple workstations is more restrictive. When an object instance is created, it is entered into the central knowledge-base making it available to all the other clients. We have not yet adequately addressed distribution of object schema or methods. We have several approaches but none have proven satisfactory for performance reasons.

The logic programming facilities of the framework are provided by LogLisp. LogLisp is a logic programming language built in, and integrated with Common Lisp. Logic is used in the framework for specifying user queries to the object base, for triggering demons, and for describing constraints on slot values. Each of these capabilities are obtained by describing a list of predicate clauses. The predicates may contain universally quantified variables.

A logic program is a collection of facts or declarations. The object base is the principle source of these facts. We are then able to write predicates or rules about how the facts are viewed. For example, suppose the knowledge base contains an instance of a project object. The project is named "Widget" and it has a manager "Simth." The object representation will be:

```
Object Class    :  Project
Object name     :  "Widget"
Object manager  :  "Smith"
```

The prediate logic representation of these facts are as follows. The "#?" symbol is a handle to the network data structure containing the instance.

```
(project #?  name "widget")
(project #?  manager "Smith")
```

Queries can easily be constructed to request all projects whose manager is "Smith" and project name is "widge" by

```
((project ?x manager "Smith")(project ?x name "widget"))
```

Integrating objects and logic is a key aspect of our logic engine. In order to perform these logic queries on the object base it is necessary to define an equivalence between the logic world and the object world. We currently do this by asserting a collection of rules describing the structure of an object its slots and its super classes. For example, given the following object definitions

```
(defobject A
   S1
   S2)

(defobject (B (:include A))
   S3)
```

From the object definitions, the following logic assertions are made by the framework

```
(A ?instance S1 ?value)
   If (A-Instance ?x)
   And(== ?value !(A-S1 ?instance))

(A ?instance S2 ?value)
   If (A-Instance ?x)
   And(== ?value !(A-S2 ?instance))

(B ?instance S3 ?value)
   If (B-Instance ?x)
   And(== ?value !(B-S3 ?instance))

(A-Instance ?instance)
   If (B-Instance ?instance)
```

Note that == is the general Loglisp equality predicate, and ! signifies that the following expression should be evaulated by Lisp rather than logic. The A-S1, A-S2, and B-S3 symbols name Lisp slot accessor functions. In addition, an A-Instance and B-Instance fact would be asserted for each instance of object class A and B respectively that are created.

The ability to use the Lisp accessor functions with the logical predicates enables us to avoid complete duplication of the knowledge base in both the logic system and the object system. Although this permits us to use the LogLisp system without change, it does require a large number of mapping predicates.

The ability to use logical queries enables us to retrieve objects using any combination of their attributes. Such a capability minimizes the need to retain complex accessing relationships to objects of interest. The logical predicates are also used to create demons and object slot assertions.

A demon associates a logical predicate with the execution of a method. Whenever the predicate is satisfied, the method is invoked. The bound variables to the logic query may be used as parameters to the method. An example of a demon is when a code object is marked as "code complete", then it should be complied. A compile demon is constructed as follows:

```
((code-object ?x status code-complete)) => (compile ?x)
```

The compile method is applied using the free variable "?x" to all objects satisfying the predicate. The assumption is that that compile method, whether it is successful or not, will side-effect the state of the code object to some state other than "code-complete". This will inhibit continual recompilation of the object.

Slot assertions similarily make use of the query mechanism to ensure satisfaction of an event. Unlike demons, assertions are intended to describe logical relationships between objects. They describe a set of values that a slot is to contain. Whenever an object is modified which may change one of the constraints, the associated predicate is evaluated to determine if a change should be made to the slot value. A principle use of slot assertions are to establish inverse relationships between objects. To do this we have introduced the "*" construct which denotes an instance of the object class being defined. For example, a project object must maintain a list of all the programmers assigned to the project. The object schema for a project will be defined as:

```
(defobject project
   (programmers '(?x)
      assert ((programmer ?x current-project *))))
```

The third use of Loglisp is to construct rules about the object base. These are predicates that use the facts already asserted. For example, we are able to define a rule stating that if an employee is not assigned to a project and is not a manager, then the employee is unassigned.

```
(defrule unassigned
   ((unassigned ?x)
      ((employee ?x manager false) (employee ?x current-project nil))))
```

Demons, queries, and assertions can now be constructed to manipulate unassigned employees.

Our initial approach to performing logic queries was to have them all evaluated on the central server machine. This permitted us to get a functional workstation query capability without having to distribute the LogLisp knowledge base across the network. A new approach has been developed to get Loglisp to directly access the distributed object base. We plan to do this by making an object schema for a LogLisp predicate object. LogLisp internally represents predicates as a Common Lisp defstruct. By modifing LogLisp to use the predicate object rather than the defstruct, we expect to be able to allow predicates to be distributed.

LogLisp clauses are represented as lists where the functor position is a symbol that names a predicate. In LogLisp, the predicate structure is placed on the symbol's property list. Since the only attribute of symbols that we distribute across the network is the symbol name, we will implement a LogLisp symbol table object containing a distributed lookup table. The lookup table relates symbols with predicate objects. In essence, we will merged the LogLisp database into the object base.

3.1 The Framework Interface

The framework interface for creating object schema, methods, rules, and demons is defined below. Brackets, braces, stars, plus signs, and vertical bars are metasyntactic marks. Brackets [and] indicate that what they enclose is optional (may appear zero times or one time in that place); the square brackets should not be written in code. Braces, { and }, simply group what they enclose, and may be followed by a star (*) indicating that what the braces enclose may appear zero or more times. Within braces or brackets, a vertical bar, |, separates mutually exclusive choices.

```
(defobject (object-name (:include (super-class)*))
    ((slot-name (value) : ((value)*)
          :type object-name
          :assert ((clause)*) )}*)

(defdemon rule-name (method) ((clause)*))

(defmeth method-name ((argument object-name)* | argument*) form*)

(defrule rule-name ((clause)*)*)
```

4 The Inspector

To display objects in the knowledge base and view the actions being performed, we have constructed a graphical inspector. The inspector contains four windows as shown in figure 4. These windows include a graph window to describe the object hierarchy in the knowledge-base, a window containing the objects slots and values, a task window for describing the demons and constraints being applied to the knowledge-base, and a Lisp Listener window. The user interacts with the graph window and the object description window to explore the contents of the knowledge-base. As operations are performed on the knowledge-base, events are triggered and displayed in the task window. The task window is for display purposes only to show the events that are occuring. The Lisp Listener window is used to run methods and modify the knowledge base.

Two additional capabilities are planned which include a multiprocess inspector and a natural language query facility. Within any single user workstation, the programmer performs many functions often in parallel. Some of these functions compete for interaction with the programmer. The multiprocess inspector will explore user interface techniques to effectively manage the screen. Particularly, with a framework that contains rules about the ordering of operations and when operations should be performed, many operations may be initiated by the framework instead of being explicitly started by the programmer. The user interface should notify the programmer of the pending activities and allow him to easily prioritize and select which task to service next. We plan to implement this through the use of multiple pop-up windows. As a new activity is started, a window will be assigned to it. When the window exposes itself to request user interaction, the programmer may choose to service the request, ignore it, or cancel the activity.

The natural language query facility will explore the construction of an inteligent help facility. Several simple natural language parsers are available to translate a natural language request into a logical query. We intend to integrate such a system into the framework and enable users to explore the project database and to obtain help on the use of the system.

5 Future Topics

The prototype we have constructed is still in its early development stages. Many topics remain to be investigated. We have already discussed some of the deficiencies in the system and our plans for resolving them. In addition, we are interested in addressing several additional topics.

5.1 Transaction Processing and Object Locking

In any multiple user database, the ability to lock objects and encapsulate transactions is necessary for data consistency. Due to the centralized implementation of the data-base, it is difficult to introduce concepts of transactions. This is particularly necessary due to the constraint checking performed by the data base.

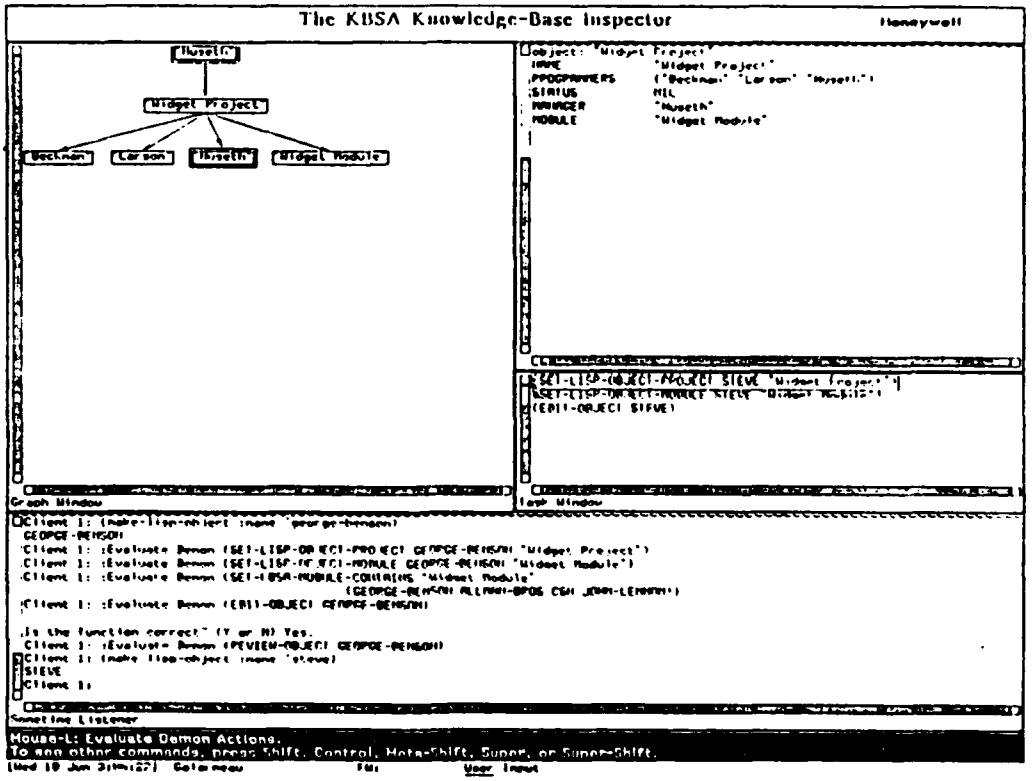


Figure 1: KBSA Knowledge-base Inspector

When a series of operations are applied to the data base, inconsistent states may exist until all operations have completed. The data base must not attempt to correct this inconsistent state until the transaction is complete.

Since multiple users are accessing the data, it is not reasonable to disable all checking of the constraints and demons. A single transaction may run for a significant length of time. One potential approach to this problem is to use object locking. Locked objects will not invoke the constraint checking mechanism when they are modified. This requires the user to lock all objects that may be potentially modified in the course of a transaction. The locked objects will not be available to any other users. The constraints will be checked when the objects are unlocked.

5.2 Access Control

As with any multiple user programming environment, a wide variety of objects and relationships exist that must be protected against accidental or malicious use. The knowledge base must be able to distinguish between users and carefully control the ability of users to examine and modify knowledge base objects. Wood80

Secure information processing technologies, particularly in distributed systems, are not well understood. Techniques remain to be developed to enable reasonable levels of data security to be maintained without excessive costs both in development of the secure system, and its resulting performance. We do not intended to address these issues other than to examine minimal levels of association between users and the objects they are responsible for and the ability to grant and deny access on those objects.

Our current approach is to control access to an object using a primitive framework object class called a *role-type*. A role object contains three slots of objects, methods, and users each with a logical assertion describing the set of instances that fill the slots. When a method is applied to an object instance, the system will be searched for a role object that contains slot values of the object instance, the method being applied, and the user making the request. The principle limiting factor of any access control system will be performance. The approach described provides a high degree of flexibility and granularity. Should performance become a limiting factor, we plan to scale back some of this functionality in favor of reasonable execution speed.

5.3 Configuration Management

Within any software development environment, the history of previous operations and instances of objects must be maintained. In order to retain such a complete history for backtracking to a previous decision or replaying the software development activity, each modification to an object must be maintained.

To manage the set of changes to a baseline object, objects are chained together in ordered lists along special relationships. These relationships are **history-of**, **alternative-of**, **part-of** and **version-of**. The **history-of** relationship links all object instances that have been modified from a baseline object. When an instance is modified, instead of changing slot values, a new instance will be created and linked to the baseline object.

The **alternative-of** relationship is distinct from the **history-of** relationship in that it identifies an *unordered* list of objects. This permits the different alternative objects to be manipulated by multiple developers and permits object histories to fork into different paths.

Objects are aggregated using the **part-of** relationship. When a new alternative of an object is created, objects of which the modified object is a part must also be changed. This will cause a cascading of new alternatives to be created by following the inverse **part-of** relationship.

The **version-of** relationship is like **alternative-of** however it is used to mark a history element of an object as being completed or ready to release. Versions will form an ordered list and therefore must be synchronized

among the multiple developers. As with the **alternative-of** relationship a new version results in propagating versions up the hierarchy.

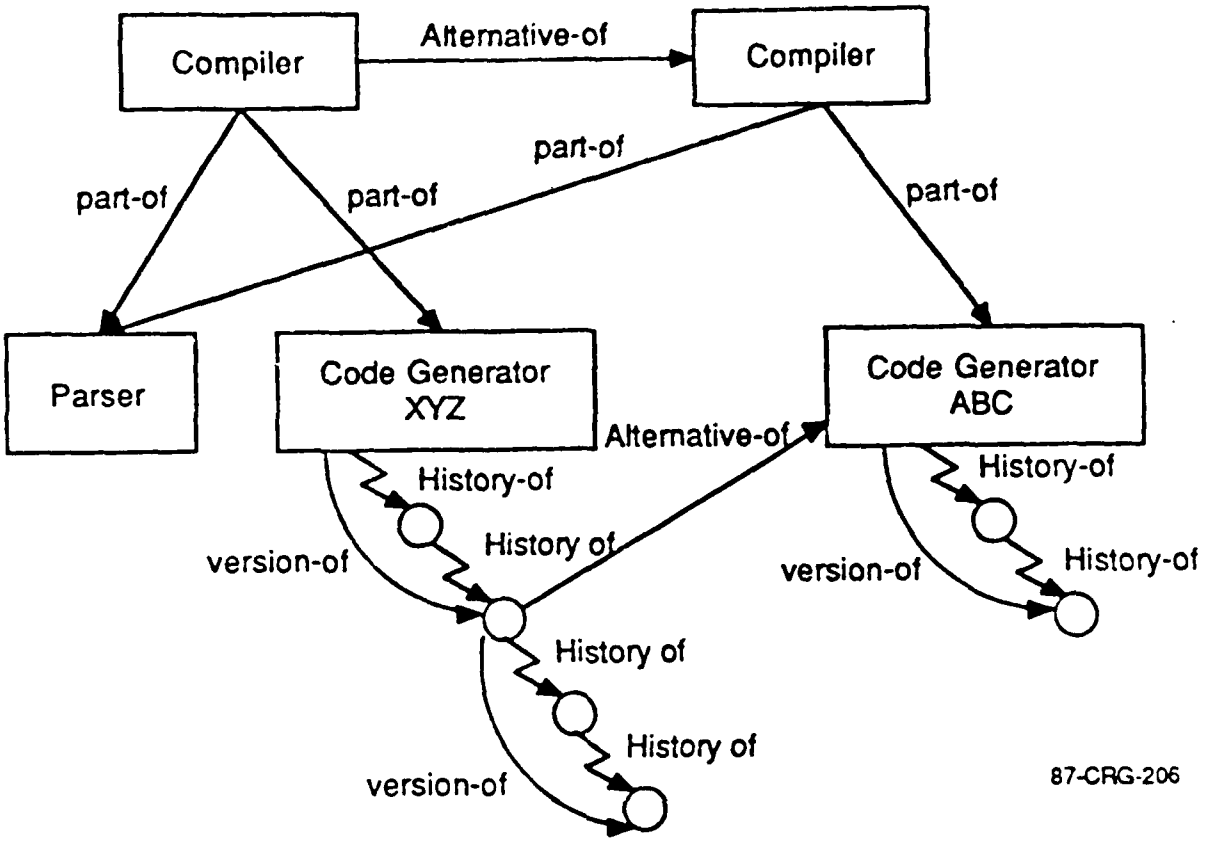


Figure 2: Configuration Management Relationships

An example of each of these capabilities is shown in figure 2 depicting a hypothetical compiler project. A compiler, targeted for the XYZ machine, is designed and its implementation started. As the code for the compiler is written, modifications are made to it resulting in new copies of the object linked through the **history-of** relationship. As the first release of the compiler approaches, it is decided to also generate code for the ABC machine. An alternative to the current code generator is created and is associated with the **alternative-of** relationship. Parallel developments are now in progress on code generators for both the XYZ and the ABC machines. Finally, the first release of the XYZ and ABC compilers result in the most recent history elements being placed in the version list using the **version-of** relationship.

References

[Bobrow85] D. Bobrow, et. al., "CommonLoops: Merging Common Lisp and Object-Oriented Programming", Xerox Palo Alto Research Center, August 1985.

[Carciofini86] J. Carciofini, T. Colburn, R. Lukat, "LogLisp Programming System Users Guide", Honeywell Systems and Research Center, July 31, 1986.

Goldberg84 A Goldberg *Smalltalk-80: Interactive Programming Environment* Addision-Wesley Reading Ma., 1984

Gupta83 A Gupta, C Forgy, "Measurements on Production Systems", Department of Computer Science, Carnegie-Mellon University, Dec. 1983

Huseth87 S Huseth, T. King, "A Common Framework for Knowledge-Based Programming", Submitted for publication

Steele84 G Steele, *Common Lisp: The Language*, Digital Press, 1984

Wood80 C Wood, E. Fernadez, and R. Summers, "Data Base Security: Requirements, Policies, and Models", IBM Systems Journal, Vol 19, No 2, 1980

Where's the Intelligence in the Intelligent Assistant for Requirements Analysis?

Inference Processes in the KBRA

Andrew J. Czuchry, Jr.
Sanders, A Lockheed Company
95 Canal Street
Nashua, NH 03061

Abstract

There are many inference processes which are supported within Sanders' prototype of an intelligent assistant for requirements acquisition [1] and analysis, the Knowledge-Based Requirements Assistant (KBRA) [2] These processes are unified in SOCLE [3], the underlying knowledge representation language in KBRA. The inference processes are the following: inheritance, automatic classification, dependency tracing, local propagation, truth maintenance, contradiction detection, default reasoning, reasonable value checks, and associative retrieval. These inference processes, combined with application knowledge, are what produce the "intelligent" behavior of this assistant. More specifically, they provide active support for the incremental formalization of requirements. Additionally, they support the following activities representative of the processes associated with incremental formalization: reuse of community knowledge, multiple viewpoints, critiquing, analysis, validation, and explanation.

The intent of this report is to explain how a variety of artificial intelligence (AI) inference techniques have been used in the context of the KBRA and the resulting "intelligent" behavior of the KBRA system. In so doing, it will indicate the inference processes we have used and how they support requirement characterization. (By requirements characterization I mean the acquisition and analysis of requirements; the declaration of ideas and the analysis and engineering of the interaction of these ideas as they are fit into a evolving model of a system to be built.) Additionally, particularly interesting examples of the manner in which inferences support the various characteristics of requirements characterization will be presented.

---

[1] Acquisition of requirements, as used here, means the declaration of ideas and their placement with some model or models of the system to be built.

[2] This work has been funded by the Rome Air Development Center at Griffiss Air Force Base in New York under Contract No. F30602-85-C-0267

[3] SOCLE, a homonym derived from Structured Object and Constraint Language Environment, Harris is a hybrid system with FRL [Roberts and Goldstein] and Constraint-based Languages [Steele] as ancestors. It supports both object-oriented and constraint-based programming styles.

1. Introduction

The current KBRA prototype is a demonstration of many of the essential features of the ultimate Knowledge-Based Software Assistant *(KBSA)* [Green, Luckham, Balzer, Cheatham, Rich] requirements assistant. It is designed to support the requirements engineer beginning at the initial phase of requirements work and continuing through to document preparation. In using KBRA, an engineer makes the commitment to working on-line not on paper. The reward for this commitment is that the decisions made and their associated justifications are not lost, but rather become part of the *KBSA* environment to be available and used throughout the entire software life cycle.

In order to achieve the goal of helping automate and mediate requirements characterization, several inference processes have been incorporated into KBRA. The inference processes are listed in figure 1a. These inference processes, combined with application knowledge, are what produce the "intelligent" behavior of this assistant. More specifically, they provide active support for requirements characterization (see figure 1b). Although this list is not exhaustive, it does cover the representative qualities of requirements characterization. The following matrix (figure 1c) highlights the inference processes within KBRA and the requirements characterization qualities which they support.

inheritance
automatic classification
dependency tracing
local propagation
truth maintenance
contradiction detection
default reasoning
reasonable value checks
associative retrieval

Figure 1a: The Inference Processes

incremental formalization
reuse of community knowledge
multiple viewpoints
critiquing
analysis/validation
explanation

Figure 1b: A Partial List of Requirements Characterization Qualities

Inference \Requirements Analysis/ Processes \Engineering Characteristics	Incremental Formalization	Reuse of Community Knowledge	Multiple Viewpoints	Critiquing	Analysis/ Validation	Explanation
Inheritance	√	√	√	√*		
Automatic Classification	*√	*√				
Dependency Tracing	√		√	√	√	*√
Local Propagation	*√	√	√		*√	√
Truth Maintenance	*√	√			√	
Contradiction Detection	√	*√		√*	√	
Default Reasoning	√	*√	√	√	√	
Reasonable Value Checks	√			√	*√	
Associative Retrieval	√	√		√		√

Figure 1c: Inference Processes and the Requirements Acquisition and Analysis Characteristics they Support

The *intent* of this matrix is to serve as a summary of the relationships between the processes and the characteristics. It is not necessarily a reference for the following report sections.

This report will initially define incremental formalization and the representatives of its associated requirements characterization qualities used in this matrix. Subsequently, it will justify the √ s in the matrix, each of which indicates that the respective inference process is used to support the indicated characteristic. Additionally, particularly interesting examples (indicated by boxes with *s in them) of the manner in which inferences support the various characteristics of requirements acquisition and analysis will be presented. As you will note, although these inference processes are called out separately, they actually are used in conjunction with each other. This intertwinement approach is no accident. The processes act as cooperating agents and produce synergistic results. The results can be seen in the "intelligent" behavior within KBRA.

2. Incremental Formalization and Requirements Characterization

Requirements characterization is the declaration of ideas and the analysis and engineering of the interaction of these ideas as they are fit into an evolving model of a system to be built. The following sections will define the qualities of requirements characterization as listed in figure 1b. The goal of this report is not to justify the inclusion of these particular processes. This information is listed and defined here to demonstrate the usage of the

inference processes in KBRA. Supporting arguments for the belief in the importance of these qualities of requirements characterizations are presented elsewhere [Balzer, Goldman, Wile], [Czuchry, Harris], [D'Addamio].

2.1. Incremental Formalization

Incremental formalization refers to the ability to capture information which is initially specified informally and to subsequently evolve that information step-by-step into a more formal specification. An example is the formalization which begins with an initial use of cryptic notes and abbreviations for a description. In an effort to extend general comprehensibility, an engineer may extend, augment, or clarify the information contained in an initial outline. In order to do this, a formal investigation of the information contained in the outline (e.g., functional flow analysis) may be initiated. As a result of formal analysis, the initially informal information becomes more formalized. This formalization may proceed automatically in some cases but most often will be accomplished as KBRA supports the engineer in performing formal analyses.

2.2. Reuse of Community Knowledge

Reusability, as used in this report, is the reuse of previously defined systems and their generalizations. It is not limited to the specification of reusable systems. This extension is important because we want to be able to modify, or at least refer to, systems which are not specifically designed to be reusable. It is supported in KBRA because engineers do this naturally when characterizing requirements.

2.3. Multiple Viewpoints

There are many ways to look at complex situations when trying to get a handle on a problem. The provision of multiple viewpoints within a system are more than a mere interface issue. It goes beyond the manifestation of views on the screen to actually addressing the needs and the processes of problem conceptualization, definition, and analysis. The types of perspectives one understands (e.g., data flow, SADT, R-Nets), when perspectives are appropriately taken (e.g., establish data and functions before the actual data flow), the types of information processed (e.g., states, data, software, hardware), and the breadth of the characteristics of information (e.g., concurrency, states only, "black box") are all factors in determining how one looks at a problem. Since different viewpoints of the same information are appropriate at different times and for different people, multiple viewpoints provided essential problem-solving elements.

2.4. Critiquing

Critiquing is the process of making indications and contraindications. The goal is to make an analyst aware of the interrelationship of requirements within the current system, comparisons with other similar systems (if they exist), the "laws" of nature, and possible contradictions with known technology limits.

2.5. Analysis/Validation

Analysis/Validation includes ensuring that the requirements which have been specified are actually those intended (e.g., the data flow meets the processing needs) as well as resolving the requirement parameters bounding the problem (e.g., the optimum size of memory given cost, physical size, response time, and program size requirements).

2.6. Explanation

Explanations answer the question "Why?". They are used to help in requirement validation by indicating requirements traceability: 1. how requirements are related, and 2. how requirements are derived.

3. Inference Processes within KBRA

In order to achieve the goal of helping automate and mediate requirements characterization, several inference processes have been incorporated into KBRA (see figure 1a). These inference processes will be described in the following sections. In so doing, particularly interesting examples of the manner in which inferences support the various characteristics of requirements acquisition and analysis, cited in the preceding section, will be presented. Although these inference processes are called out separately, they actually are used in conjunction with each other. This intertwinement approach is no accident. The processes act as cooperating agents and produce synergistic results. The results can be seen in the "intelligent" behavior within KBRA.

3.1. Inheritance

Incremental formalization takes advantage of inheritance. As requirements become more formalized, (e.g., an uninterpreted requirement [in KBRA the most informal type of a requirement] becomes an activity; data becomes geometric data) information in addition to that which is explicitly specified is inferred. An example of such additional information is an expectation about data within an activity, whereas no such expectation could be created for an uninterpreted requirement since the uninterpreted requirement may itself turn out to refer to data flow rather than an activity. Inheritance is used to retrieve expectations and properties of objects as they become more formalized. Figure 2 indicates the expectations associated with activities and the lack of expectations for uninterpreted requirements. A component of formalization thus becomes the determination of the taxonomic relation (class relation) of requirements within the general hierarchy of requirements knowledge.

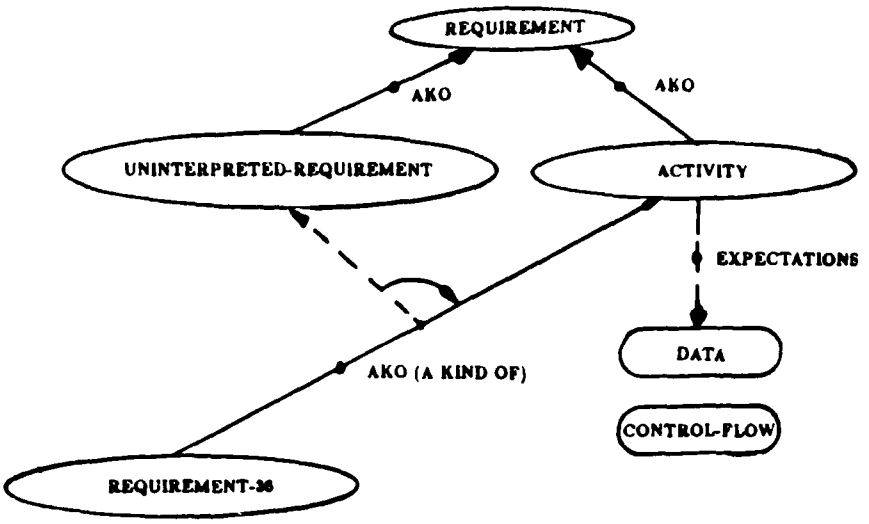


Figure 2: Inheritance and Expectations within Requirements Knowledge

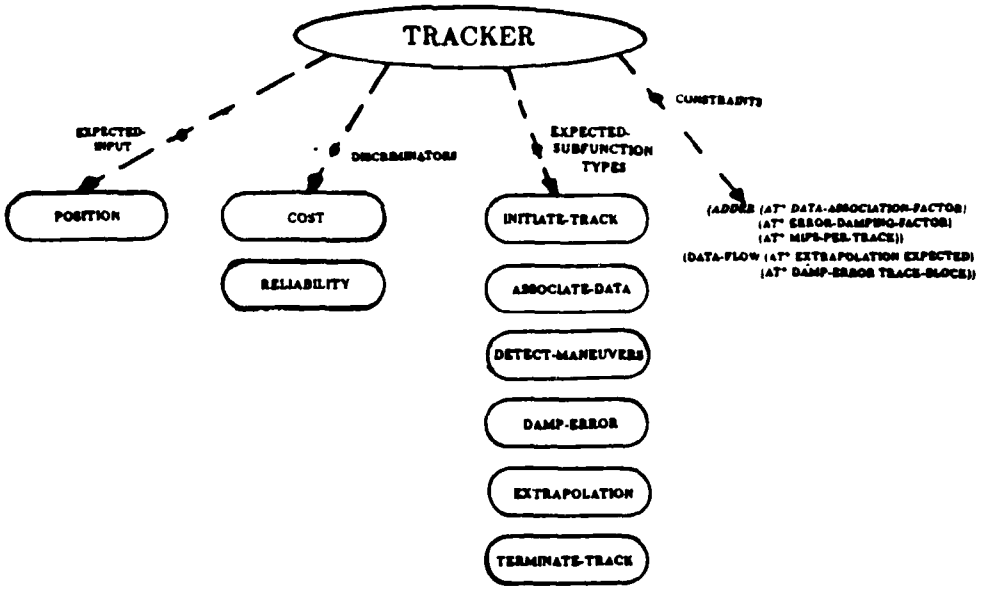


Figure 3: Class Frames for a Tracker used in an Air Traffic Control System.

Inheritance can also be exploited in addressing reusability. Class frames[4] ('generic frames') can be used to represent the generalizations of concepts. These 'generic frames', and thus generalizations for solutions to requirements problems, are available in a library created by knowledge engineers.[5] An example is presented in figure 3. Reusability becomes a matter

[4]Class frames may be thought of as active templates of distinguishing features and properties of the class its members, and its subclasses.

[5]Inheritance facilitates knowledge engineering because it allows concentration on the details of what makes each class different since only that distinguishing information needs to be explicitly recorded. The rest of the characteristics can be inferred (inherited) from superclasses.

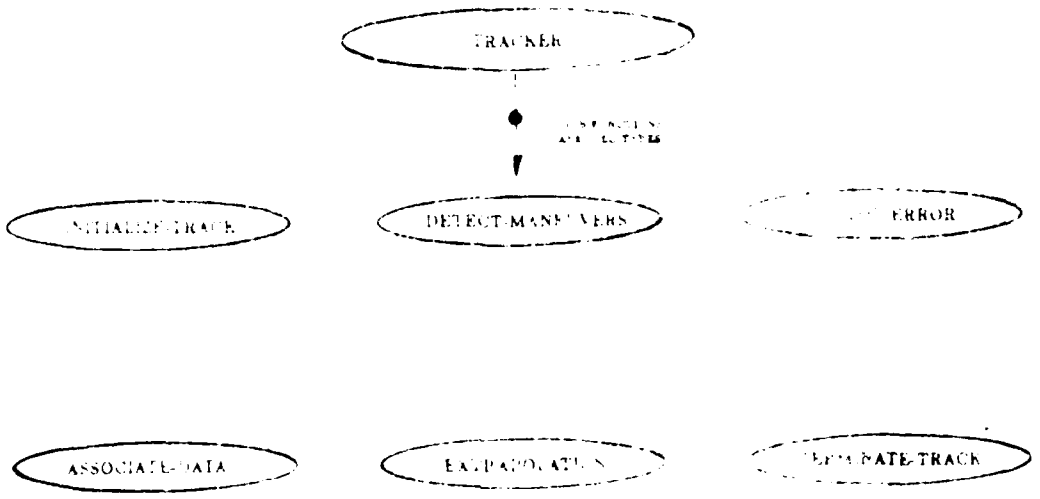


Figure 4a. A Generic Tracker (from the ATC application knowledge module)

of instantiation of the generic frames. Since frames can contain explications, constraints, relationships, and structure, instantiation results in a full-bodied concept (see figure 3) which forms the foundation for reuse

Inheritance is used within the presentations which create Multiple Viewpoints. The information that is presented in a Review presentation varies dependent upon the classification of a requirements entity. For example, the processors and consumers of data items are presented while the subfunctions, data, and [illegible] are presented for activities.

Critiquing takes advantage of inheritance in a slightly different manner. It uses inheritance to compare the requirements as defined by the engineer with generalizations for known components and systems stored within the community knowledge data base (application knowledge modules). Through the act of specifying a system to be defined as an Air Traffic Control System (ATCS), comparisons (such as those described below) are made between the generic frame for ATCs (and its components) and the system which the engineer is defining. This is used to suggest possible inconsistencies or incompleteness.

For example, a generic tracker is depicted in figure 4a. Since this tracker appears in the community knowledge of the ATC application knowledge module, it can be used for comparison to any tracker specified by an engineer (e.g., figure 4b). KBRA can critique the tracker requirements and give the engineer the following feedback: "Most trackers have a Terminate-Track requirement, but you seem to be missing one." This critique is the result of a comparison between the expected subfunction types on the class frame and the types of the actual subfunctions for the new tracker

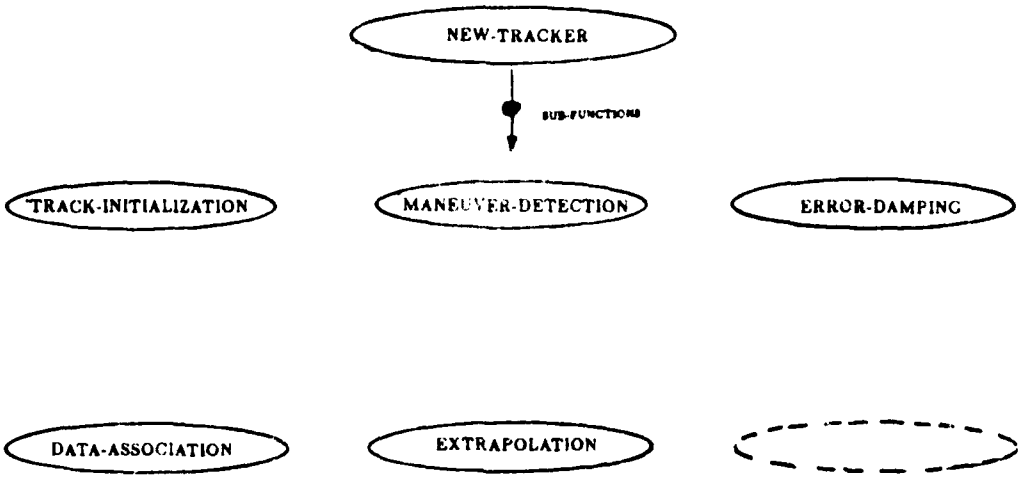


Figure 4b: A Tracker being specified while establishing the requirements for an ATC

3.2. Automatic Classification

Automatic classification is the process by which objects are automatically placed in the most specialized class, within the requirements knowledge hierarchy, consistent with what is known about an object. This is important because specialized classes contain more detail and expectations than their superclasses. As a result, additional information can be deduced (through inheritance as mentioned earlier) when classification takes place. Although KBRA does not have a general purpose classifier, application specific automatic classification is supported. The classification proceeds using special purpose classification decision trees which exist in KBRA application knowledge modules.

Automatic classification takes place throughout the incremental formalization process. For example, when a source or sink requirement entity is added to a datum, that requirement entity (which may have been uninterpreted up to this point) is automatically classified as an activity.

Automatic classifications also play a role when components are reused. The following is an example. Upon indication of a tracker as a component of an Air Traffic Control system (ATC), KBRA will classify the tracker to the best of its ability using the decision trees cached on the generic tracker frame. It currently knows about several different types of trackers (e.g., kalman filter, alpha-beta, alpha-beta-gamma, multiple hypothesis) each of which have different distinguishing characteristics. If there is a need for a high degree of accuracy and reliability and not a stringent requirement on processing time, the system will classify the tracker as a kalman filter. Since related requirements imply certain other

requirements (in the situation indicated above, a kalman filter is act[illegible] other requirements), it is important to have the system automatically [illegible] those implications and take them into account when requirements are [illegible]. In addition to recognizing implied requirements by successful automatic classification, the feasibility of related requirements are checked. In cases where classification is not possible (e.g., high accuracy, high reliability, fast processing time), requirements inconsistent with current technology are recognized.

3.3. Dependency Tracing

Dependency tracing is fundamental to the KBRA. It forms the foundation for several inference processes: local propagation, truth maintenance, and contradiction detection. Each of these processes trace dependencies in order to perform their inferences. Additionally, dependency tracing supports requirements characterization independently of these other inference processes.

It plays a role in helping support the formalization of requirements, presenting requirements from many viewpoints, critiquing choices, analyzing and validating specifications, and explanations. Of all of these, it is especially important for explanations. The "whys" of requirements are associated with dependency links which must be traversed. The goal of this traversal is to ascertain the premises from which the requirement has been derived and the relations between those premises and the requirement in question. The search space of requirements entities is managed by searching among only those entities with dependency relations. This addresses the needs which occur during requirement traceability.

An example of dependency tracing is the ability to trace "because" relations. A certain requirement for processing time may be set "because" there is a requirement for near-real-time performance. As long as the requirement for near-real-time performance is valid, the processing time will persist. If the requirement for near-real-time performance is retracted, however, there may no longer be any support for the processing time. If this is the case, the processing time will be retracted. (A default processing time will be reasserted, if possible, through default reasoning.)

3.4. Local Propagation

There are two classes of local propagation: 1. dæmons or procedural attachments, and 2. constraint propagation.

An example of local propagation using procedural attachments is the completion of partial descriptions (an automated part of incremental formalization). When requirements for new activities are stated, default data may exist (see default reasoning, section 3.7). There may be some way to "wire-up" this dangling data (e.g., existing solely as input to the new activity) to other data which is already present in the system (e.g., the output of another activity). The use of the X and Y values from a stereographic projection of a radar return

could be used as the location [illegible] tracker. The [illegible] proceeds automatically by triggering the procedural attachments stored in each slot.

An example of constraint propagation [illegible] and validation. The specification of an equation relating requirements [illegible] the following example propagation occurs due to the declaration of a simple [illegible] formula in an air traffic control system. The context of this formula is [illegible] Figure 5a.

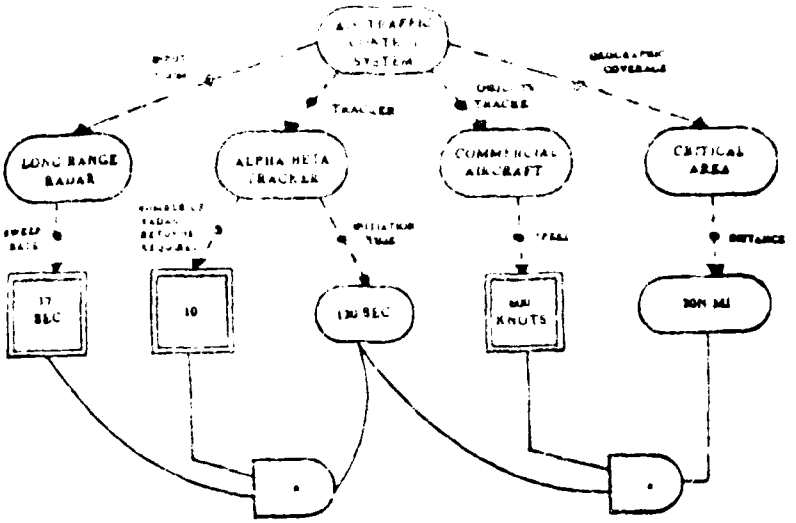


Figure 5a. Propagation of Values within the Structured-Object Decomposition for the AIR TRAFFIC CONTROL SYSTEM Concept

This figure shows a partial decomposition of the AIR TRAFFIC CONTROL SYSTEM using "input-from", "tracker", "objects-tracked", and "geographic-coverage" slots. The slot fillers for each of these parts are LONG-RANGE-RADAR, ALPHA-BETA-TRACKER, COMMERCIAL-AIRCRAFT, and CRITICAL-AREA respectively. Each of these is a structured object which (as noted in the section on inheritance) inherits values, defaults, and procedural attachments from generalized concepts.

In such a decomposition, slots which are related through formulas may be functionally far apart. For example, the distance that an aircraft can cover before a track is established is related to the speed of the aircraft, sweep rate of the radar, and the number of hits required for the tracker to establish a new track. Letting

R = the sweep rate of the system radar
N = number of radar returns required to establish a new track
T = initiation time for establishing a track
S = speed of a commercial aircraft
D = distance covered by the aircraft before a track is established

we can quickly establish the formulas:

$$R * N = T$$
$$T * S = D.$$

These formulas are illustrated on the diagram as wiring networks. When values are set for "sweep-rate", "number-of-radar-returns-required", and "speed" as shown (12 sec, 5, 600 knots respectively), then values of 1 minute (60 sec), and 10 nautical miles are automatically propagated to the "initiation-time" and "distance" variables, and need not be entered as independent requirements.

These equations propagate values in a bi-directional manner. Figure 5b depicts the "reverse" propagation. Values are set for the "sweep-rate", "speed", and "distance" (12 sec, 600 knots, and 10 nautical miles respectively). The calculated values 5 and 1 minute (60 sec) appear for the "number-of-radar-returns" and "initiation-time" respectively.

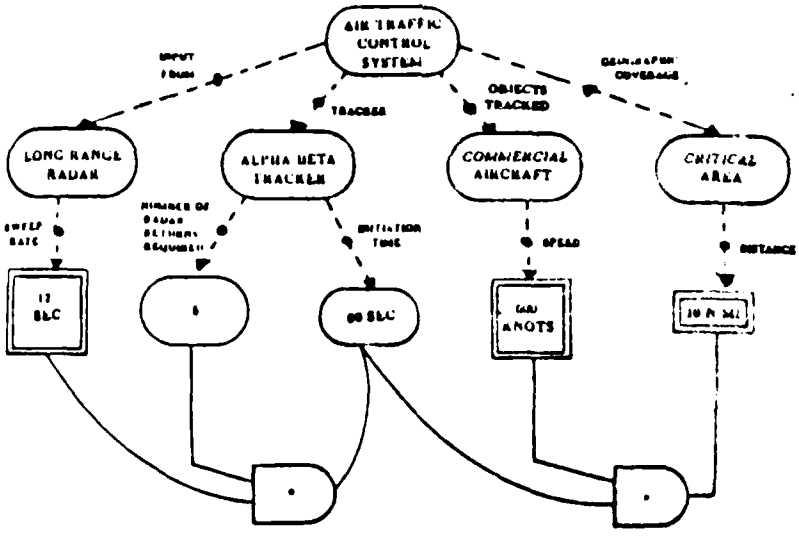


Figure 5b: "Reverse" Constraint Propagation

3.5. Truth Maintenance

Truth maintenance supports changing your mind. Changing your mind could occur through out incremental formalization, reuse, or analysis validation. Truth maintenance in KBRA is realized through constraint maintenance and constraint propagation.

Constraints are only enforced [illegible] structured-object networks. Subsequent [illegible] structured-object networks, and constraint [illegible]. The adjustment of constraint networks is [illegible] of hierarchical structure and is an example of [illegible] truth maintenance. The result is that [illegible] are dependent on the value retracted are revisited [illegible] essary to replay all the equations in the system [illegible]. For example (figure 6a), if the "input-from" [illegible] second value will be disconnected from the formula [illegible] and was a premise will be retracted (in the current [illegible] point time" and 10 nautical mile "distance" [illegible] LONG RANGE RADAR in figure 6c), the constraint [illegible] value. The fact that a new value of 3 seconds [illegible] constraint network. Appropriate [illegible] propagated (see local propagation section 3.4). This support [illegible] of formulas between values only as they fill slots in structured-object networks is a key feature in KBRA

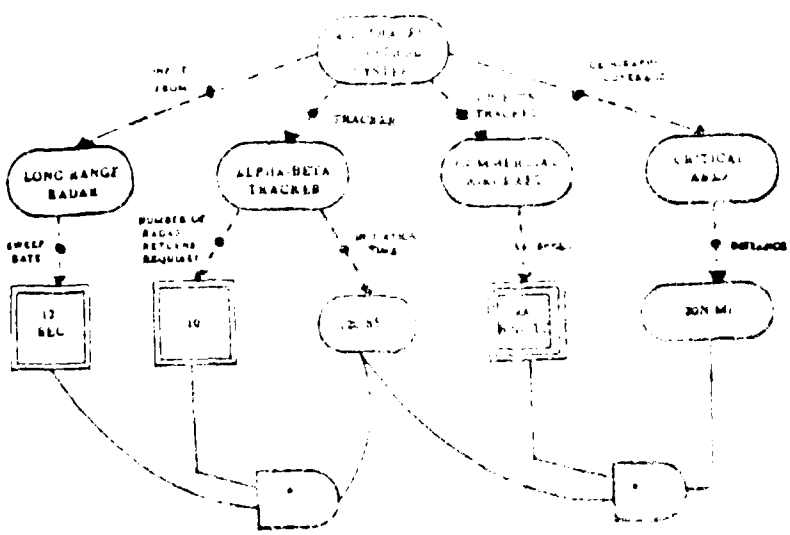


Figure 6c. [illegible]

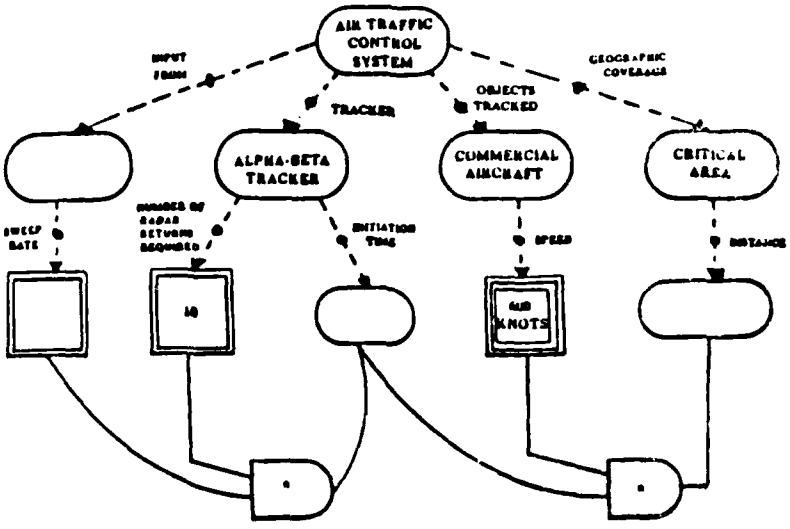


Figure 6b: Constraint Maintenance upon the Retraction of a Structured-Object

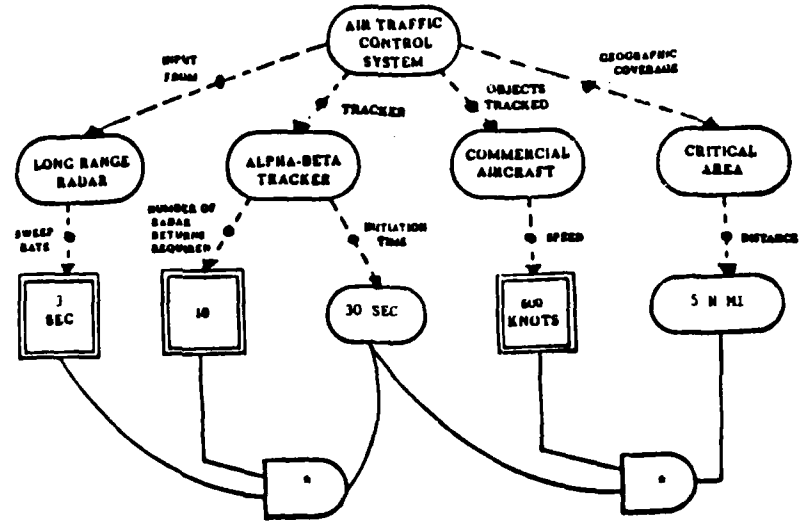


Figure 6c. Constraint Maintenance upon the Assertion of a Structured-Object Slot Filler

3.6. Contradiction Detection

Contradiction detection provides feedback when inconsistencies are determined. These inconsistencies could be in the form of incompatible values and relations (e.g., a case where X implies not Y, X is true, Y is true) detected through unit propositional resolution or in the form of type errors (e.g., using an ACTIVITY as INPUT into another ACTIVITY [only DATA can be an input]). Contradictions often arise in a variety of circumstances. They arise as information is formalized. When this happens, feedback must be given so that compatible requirements may be specified throughout each component of the system. Analysis/Validation is partially an attempt to resolve numeric completeness, and type expectation contradictions. The use of contradiction detection in the reuse of community knowledge and in critiquing will be discussed in greater detail.

Contradiction detection is important for reusable components because it traps conditions when a component should NOT be reused. Examples of such conditions are as follows: 1. magnitude discrepancies (e.g., response time of 6 sec is possible using a certain display, but a response time of 0.5 sec is required. In this case, a display with the appropriate response time should be chosen), 2. type inconsistencies (e.g., one compound expects a location as polar coordinates for an input, but the input will be X,Y coordinates. In this case, a different component should be used: one that expects X,Y coordinates).

It is especially critical in critiquing. If possible, KBRA will automatically resolve the inconsistency based on the level of support indicated for the requirements involved. The declaration of a level of support is available to the user. Included are default (use only when nothing else is available), supposition (try out a possible value), belief (this value is very probable), and constant (reserved for definite constants e.g., π). The most tentative level of support among the inconsistent values is automatically retracted. If more than one value has the same level of support, the user is prompted to make a selection. As an example, let us return to the AIR TRAFFIC CONTROL SYSTEM. The values depicted in figure 7a are all consistent. However, if there is a requirement that the critical area be 10 nautical miles, then a contradiction will arise (see figure 7b). One of the requirements that has been explicitly set ("sweep-rate", "number-of-radar-returns", "speed", "distance") must be retracted.

If there were only one premise at the lowest level of support (e.g., if "number-of-radar-returns" were a supposition while the others were beliefs), it would be retracted. However, in the example depicted in figure 7b, the "sweep-rate", "number-of-radar-returns" and "distance" have been entered as suppositions, while the "speed" has been entered as a belief. Since it is not possible to automatically resolve the contradiction (there are three premises at the weakest support level), the user will be prompted to make the resolution. KBRA will help focus attention on the appropriate choices by presenting only the premises with the lowest level of support, in this example the suppositions ("sweep-rate", "number-of-

radar-returns", and "distance". Note that the belief "speed" is not presented for retraction. If the user chooses to retract the "number-of-radar-returns", or if it were automatically retracted because it was the only premise at the weakest support level, then a value consistent with all the other premises will be propagated to it (value 5 in figure 7c).

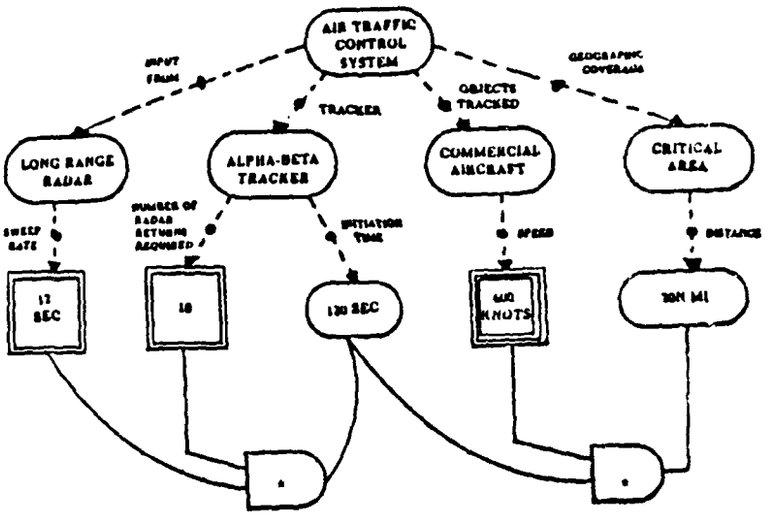


Figure 7a: Consistent Premises

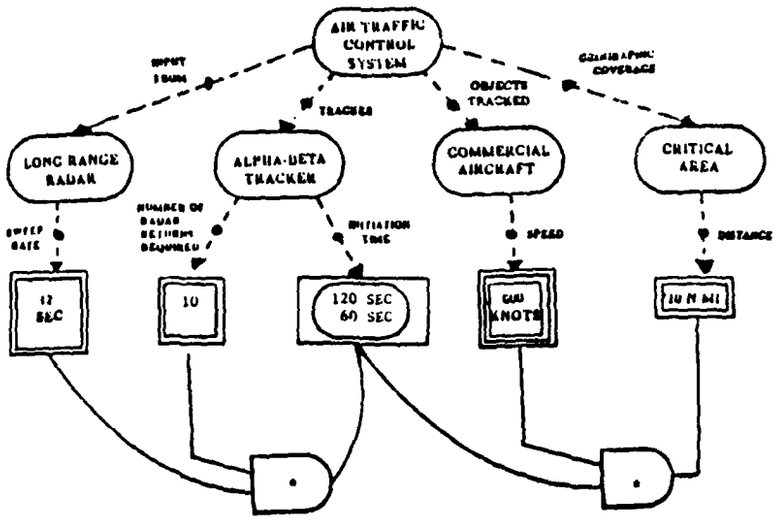


Figure 7b: Contradictory Premises

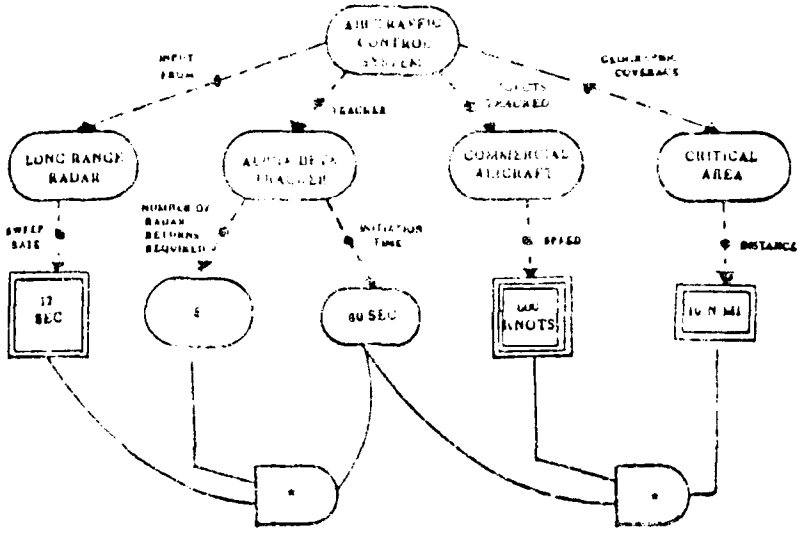


Figure 7c: A New Set of Consistent Premises

3.7. Default Reasoning

Default reasoning is an important issue [Reiter]. For us, it encompasses: 1. the existence of default values, and 2. the persistence of default values (they can be overridden, but reassert themselves when the overriding values are retracted). This type of reasoning is used in all aspects of the KBRA. When nothing else is known, this reasoning is used to determine a plausible response, characteristic, or value.

An example of this occurs in reusable components. For most activities, there is an expectation of data flow. In the specific case of a tracker, a location input is expected. Location thus appears as a default input for a tracker. By incorporating the tracker into a system which processes radar returns to compute rectangular coordinates (e.g., stereographic projection), the actual inputs to the tracker would be the X and Y values rather than one "location". The location input would thus bow out and the new inputs would be the X and Y values. If the coordinate conversion of the tracker is removed (retracted) at some point in the future, the default expectation of a location datum input into the tracker would be reasserted. The internal representation of this is depicted in figures 8a, b, c.

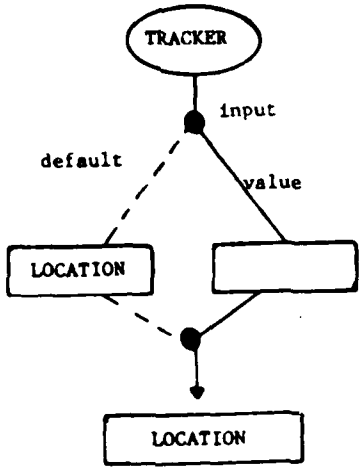


Figure 8a: A Function with Default Tracker Input of Location

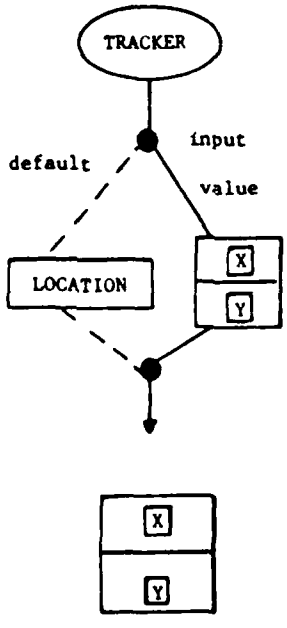


Figure 8b: A Function with Default Input Overridden

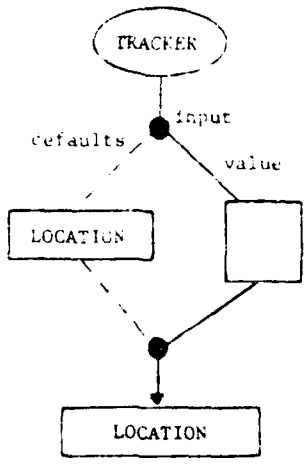


Figure 8c: A Function with the Default Input Reasserted

3.8. Reasonable Value Checks

Most object-oriented systems contain a provision for reasonable value checks. It is, therefore, not revolutionary that we take advantage of this in KBRA. However, it is important to indicate the wide range of reasonable values that can be specified in this system. Predicates can be attached to slots of frames. When values are entered, the associated reasonable value predicates are evaluated to determine if the value is reasonable. Since such predicates can be any Lisp form, a wide spectrum of reasonability testing is supported.

We take particular advantage of reasonable value checks in incremental formalization, critiquing, and analysis/validation. They are used in incremental formalization to indicate reasonable manners in which requirements may be formalized (e.g., reasonable performance characteristics). It plays a role in critiquing because the system can acknowledge magnitude errors (e.g., a screen line width of 3 inches instead of 0.03 inches) as well as technological limits (e.g., 15 MIPS for a certain machine by the year 1988).

Reasonable values also play a role in analysis and validation for catching errors implied by computed values or by verifying that values fall within acceptable limits. For example, while analyzing the response time, processing time per unit, and throughput for a tracker, a value of 10 pico seconds is calculated for the unit processing time (figure 9). This could have been calculated from the response time (10 nano seconds) divided by the

throughput (1,000). The value of 10 pico seconds for the unit processing time is clearly not within the current technology (nano seconds are about the current minimum) so this "unreasonableness" will be critiqued by KBRA (see critiquing, section 2.4).

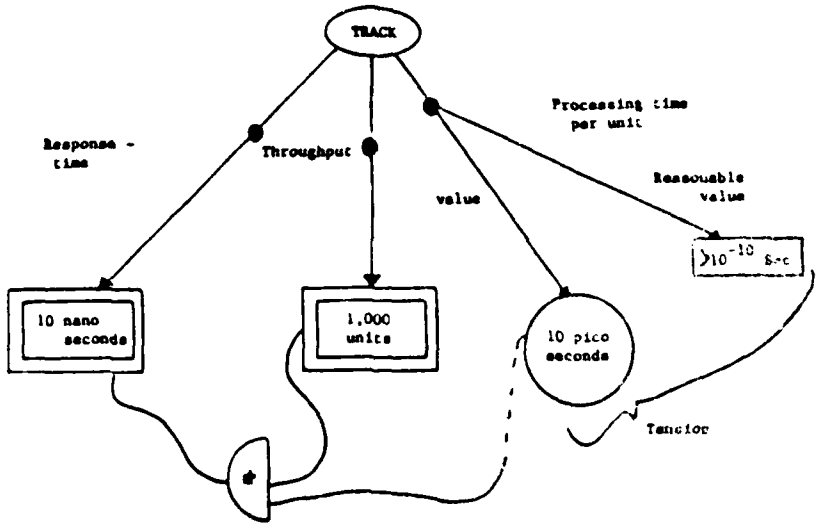


Figure 9: Reasonable Value Checks on Calculated Values

3.9. Associative Retrieval

The main importance of associative retrieval is the provision of indexing so that information is retrieved by relations rather than by searching the entire space of requirements entities. Managing the "search space" is a problem for both expert systems and intelligent assistants. Associative retrieval helps manage the "search problem" by searching only through related requirements entities (KBRA's search space), rather than all the requirements entities that may exist in the entire system. Requirements may be "related" in a variety of ways (e.g., subfunction, superfunction; data and its accessors; a name and things with that name). Further search control is provided by limiting the search to only one specific "relation" (e.g., callers of a function; consumers of a datum).

Associative Retrieval provides indexing throughout KBRA. It aids reusability by providing controlled access into a library of system components. This helps overcome one problem with reusable components: knowing the components which are available. Associative retrieval supports the engineer in incremental formalization when s/he is "getting a handle" on the formalization process. KBRA determines the answer and responds to inquiries into where a certain datum is used, who is its producer, who consumes it, what text strings mention the datum's name, what other things have the same name, what names

are synonyms, etc. Critiquing can exploit associative retrieval to ascertain the respective roles of requirement entities. Associative retrieval is used in Explanation by ascertaining the role something is playing in the system and its relationships to other requirements.

Associative retrieval is a means for determining not only that things **are** related but also **how** they are related. This is because the indices provide a path of concept-role (frame-slot) relationships. In fact, associative retrieval is fundamental to all the inference processes in KBRA. It is impossible to infer something if there is no relation established. The other KBRA inference processes (figure 1a) take advantage of the associations provided by general indexing and associative retrieval.

4. Limitations and Possible Extensions to Current Capabilities

This section to elucidate four possible extensions to the limitations in the current KBRA inference processes. They have been chosen because I have some particularly interesting ideas on these extensions. They have been motivated by our observations of requirements-level work.

Our current work on KBRA in support has focussed on two extremes of the **initiation** of requirements characterization: 1. specifying a completely new system which has never been done before (e.g., SDI) or 2. specifying the "next generation" of a system (e.g., the "next generation" ATC systems). We have not focussed on incorporating an old system of one type (e.g., patient monitoring) as an example of the characteristics for a different type of system (e.g., aircraft monitoring, ATC) [Johnson]. The possible KBRA extensions in sections 4.1 and 4.2 address this issue. These extensions are admittedly hard problems, but their investigation may yield fruitful results.

4.1 Automatic Generalization

Currently, all generalization frames must be explicitly created. In the future, generalization frames could be created by the system itself in the form of abstraction transformations (extracting common features between various frames throughout the system to make a new superclass). This would particularly aid reusability as new reusable components could be inferred and then instantiated.

4.2. Reasoning by Analogy

There is currently no reasoning by analogy in KBRA. It would, however, provide a useful extension. Associative retrieval could be pushed to reasoning by analogy; associations could be inferred by abstraction transformations. Abstraction transformations could deduce commonalities between entities and thus, create a description at a higher-level of abstraction. This new description would connect multiple entities which would previously have been unlinked. The new descriptions could be incorporated into the community knowledge and could play a role in reusability.

4.3. Judgments

There is currently no provision for making judgments based on the roles related objects play on their respective entities. Associative retrieval could be used as the foundation for such judgments. In Critiquing, the following may occur: an entity plays a certain role and is associated to other entities which play various respective roles in the system. A "judgment" could then be made on these relationships. An example is that a Kalman Filter should not be used as the tracker filter when military aircraft are used as objects to be tracked. This is because the use of military aircraft as the objects tracked precludes the use of a Kalman Filter as the track filter since a Kalman Filter is based on the assumption that the objects being tracked are not maneuvering.

4.4. General Purpose Automatic Classification

Since there currently is not a general purpose classifier as in NIKL [Kaczmarek, Bates, Robins] or KL-ONE [Schmulze & Lipkis], the addition of such a classifier would extend the current classification processes.

5. Conclusion

There are many inference processes within KBRA: inheritance, automatic classification, dependency tracing, local propagation, truth maintenance, contradiction detection, default reasoning, reasonable value checks, and associative retrieval. These inference processes are combined to actively support the acquisition and analysis of requirements; they support the incremental formalization of requirements by providing desirable "intelligent behavior" within the KBRA system. Thus, the inference processes, in conjunction with application knowledge, are where the intelligence of the intelligent assistant for requirements is found

Acknowledgments

This paper presents the work that David Harris and I have been performing over the last year and a half. Its content has been significantly influenced by David Harris and also by Dr. Charles Rich of MIT. Valuable constructive criticism has been provided by Patricia Ahern and J. Terry Ginn. My ideas regarding abstraction transformations were initiated by Lewis Johnson of ISI. The manuscript has been carefully prepared through the dedicated efforts of Lorraine Roy.

References

Balzer, Goldman, Wile, "Informality in Program Specifications", IEEE Transactions on Software Engineering, Vol. 4, No. 2, March 1978.

Czuchry, Harris, "Progress Report on a Knowledge-Based Requirements Assistant", Proceedings of the First Annual Knowledge-Based Software Assistant Conference, Rome Air Development Center, 1986.

D'Addamio, "Sanders' Artificial Intelligence Technical Report on Air Traffic Control", Sanders Associates, March 1987.

Green, Luckham, Balzer, Cheatham, Rich, "Report on a Knowledge-Based Software Assistant", Rome Air Development Center, Technical Report RADC-TR-83-195, August 1983.

Harris, "A Hybrid Structured-Object and Constraint Representation Language", Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, PA, 1986.

Johnson, W. Lewis, ISI, Personal Cummunication

Kaczmarek, Bates, Robins, "Recent Developments in NIKL", Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, PA, 1986.

Reiter, "A Logic for Default Reasoning", Artificial Intelligence **13** (1980), pp. 81-132, North Holland Publishing Company, 1980.

Roberts, Goldstein, "The FRL Primer", MIT AI Memo 408, 1977

Schmulze & Lipkis, "Classification in the KL-ONE Knowledge Representation System", Proceedings of 8th International Joint Conference on Artificial Intelligence, 1983.

Shrobe, "Dependency-Directed Reasoning for Complex Program Understanding", PhD Thesis, MIT AI TR-503, April 1979.

Steele, "The Definition and Implementation of a Computer Programming Language Based on Constraints", PhD Thesis, MIT AI TR-595, August 1980.

Turning Ideas into Specifications

W. Lewis Johnson
USC/Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292

Abstract

Specification-based software development makes software easier to validate and maintain. Yet specifications of large systems are themselves large, making understanding and validation difficult. The specifier usually must meet a number of *goals* and *requirements* at once in a specification. Each goal or requirement may be straightforward in isolation, but the underlying simplicity gets lost as interactions between requirements are discovered and tradeoffs are made. In the Knowledge-Based Specification Assistant project, we focus on the steps that specifiers go though in developing detailed specifications from initial ideas of what the specification is supposed to do. The Specification Assistant aids in developing, evaluating, and explaining specifications. This paper will discuss the development aids in the Specification Assistant, and show how they help in the transformation of requirements into specifications.

1. Introduction

The goal of automatic programming is to allow people to describe what they want a computer program to do, and have the program be automatically generated on the basis of this description. It has been generally assumed that the program description would take the form of a formal specification [4, 15]. But this raises a fundamental question: how do you know that the formal specification really describes the computer system that you have in mind? Formal specifications are hard to read, and their semantics do not always agree with one's intuitions. It is necessary to ensure that the specification is validated before it is used as the basis for implementation.

One way of making specification validation easier is to make the specification fit the way people think about systems as closely as possible [2]. In other words, the specifier should make sure that the terms used in the specification correspond well to the terms that people typically use in thinking about the domain. Another answer is to provide tools which assist people in interpreting a specification. For example, symbolic evaluators can be used to execute the specification and look for unintended behavior [5, 6]. Other analysis tools can examine the output traces of the symbolic evaluator, and point out potential problems to the specifier [19]. If the specification is paraphrased into some other medium, such as English, then the new perspective on the specification can lead to the discovery of specification bugs. [18].

Such techniques are all useful, and we continue to make use of them in our work with specifications at ISI. However, they are not sufficient when one is developing a specification for a large, complex system. The problem is that a non-trivial computer system must usually meet a number of requirements. Some of these describe normal-case behavior; others deal with exceptional cases. One can easily get lost in all the details. Furthermore, the goals that a system must meet may conflict. The specifier must make tradeoffs among the goals, or reformulate the goals into goals which do not conflict. The various system components may be designed to accomplish some overall system goal which is never explicitly stated. In general, specifiers make numerous design decisions before they start writing the specifications. As a result, it's hard to tell how the requirements stated in a typical specification relate to the real requirements

that a system must meet. If the requirements of the system change during the maintenance phase, it becomes to difficult to decide how to reconcile the new requirements into the the existing specification.

In order to make specifications really useful and easy to understand and validate, we must recognize that specification development is a problem-solving process. We can then try to build automated tools that assist in that process. For example, we can encourage specifiers to enter into the system their initial ideas about what a system should do. The system can then help the specifiers to refine and integrate those requirements, and compromise them where necessary. It can keep a record of the refinement steps, so that they resulting specification can be understood in terms of high-level requirements. The result will be a specification that is more easily validated, more easily understandable and explainable, and hence more maintainable. That is what we are developing in the Knowledge-Based Specification Assistant project.

2. Development Assistance in KBSA

The Knowledge-Based Specification Assistant project is part of a larger effort, the Knowledge-Based Software Assistant Project [3]. An overview of the Knowledge-Based Specification Assistant Project can be found in two other documents, both published in this volume: "Overview of the Knowledge-Based Specification Assistant", and "Demonstration of the Knowledge-Based Specification Assistant". We will focus here on the methodology of specification development that KBSA supports, and some of the more advanced techniques that KBSA will use to support this methodology.

In the initial stages, the specifier enters a set of requirements, using the Knowledge-Based Requirements Assistant [17]. The Knowledge-Based Requirements Assistant, or KBRA, is a system being developed by Sanders Associates. These requirements can take a variety of forms, some formal, and some in natural language. It then passes a general model of the domain, together with the system requirements, to KBSA. The Specification Assistant translates the Requirements Assistant's domain model into ISI's specification language, Gist [13, 14]. The system requirements are not automatically translated into Gist, because such a translation is likely to reveal ambiguities in the

requirements that would have to be [illegible] by the specifier. Instead, the specifier must browse through the requirements, and manually render each one into Gist.

Once the requirements are entered in the Specification Assistant, the specifier crafts a specification meeting the requirements, with help from the Assistant. During this process, two activities take place.

- The specifier identifies the most essential high-level statements from among the stated requirements, and builds a *high-level specification* of the system. The high-level specification outlines the principal goals and requirements that the system ought to achieve. It may not be possible for the implemented system to achieve these goals exactly, or in all cases. Such pragmatic considerations are not important at the high level, however. The purpose of the high-level specification is to formalize the requirements as straightforwardly as possible, so that they can serve as the basis for further development.

- The specification is gradually elaborated and refined, to take into account the pending specification tasks. Simplifying assumptions in early versions of the specification are gradually weakened. Exceptional cases are described. Operational procedures for achieving system goals are defined. The final product is referred to as the *low-level specification*. KBSA keeps track of the refinements that are made, so that low-level specifications can be understood by relating them to the high-level specifications.

KBSA is able to provide the most direct assistance in the elaboration and refinement process, in getting from high-level specification to low-level specification. The principal mechanism in KBSA for performing such elaborations and refinements is the *high-level editing command*. High-level editing commands are similar to the transformations used in transformational implementation systems [9]. Unlike conventional transformations, however, high-level editing commands need not be correctness-preserving. Their principal function is not to preserve specification semantics, but to modify it in some desired fashion. Each editing command is described in terms of its effects on the meaning of the specification, not on the superficial syntactic changes that result. They embody knowledge about how to direct such functionality changes in a specification. The specifier can concentrate on what functionality he or she wants, and how the functionality described so far needs to be refined. The high-level editing commands then carry out the necessary changes. Although the user currently must select which

high-level editing command to apply at each stage, we are currently addressing the question of how to automatically provide assistance to the user in selecting which command to apply.

3. An Example

Our discussion of KBSA's development assistance will focus on a particular specification problem, an air-traffic control system.[1] The air-traffic control system is supposed to assist controllers in tracking and controlling aircraft throughout an air space. Each aircraft is presumed to have filed a flight plan. The job of the controllers is to ensure that the aircraft adhere to the flight plans. During an aircraft's flight it may travel through multiple air spaces, each controlled by a different air traffic control facility. As the aircraft move from air space to air space, control of the aircraft must be handed off from one facility to the next. Within a given facility individual controllers may also hand control off to each other.

A high-level description of a problem, such as the one in the previous paragraph, can easily get buried in a detailed specification of a system. This is true for a number of reasons.

- High-level goals of the system are achieved indirectly by multiple low-level system functions. For example, in the air-traffic control system requirements document that we studied, the system was required to blink the display of flight plans that have not been explicitly controlled by a controller. This indirectly ensures that aircraft stay on their flight path, since assigning a controller to the flight is a prerequisite for ensuring that it stays on course.

- High-level goals may need to be compromised in order to comply with properties of the environment. For example, aircraft positions are monitored by radar. There is a margin or error inherent in these position measurements. As a result, radar tracks can get lost, and mismatches between aircraft and radar tracks can occur.

- Different system requirements can interact with each other. For instance, a

---

[1] This problem is one of several that we have studied in the context of the project. Our principal focus has been on two problems, however: the air-traffic control problem and a hospital information systems problem.

function for changing flight plans must be included in the air traffic control system. Such a facility can interact with facilities for handing off aircraft: if a flight plan changes, it may have to be handed off to a different controller.

4. Going from High-Level to Low-Level Specifications

In what follows, we will look in detail at some of the properties of high-level specifications, and discuss how as they are transformed into low-level specifications. This will illustrate how the complexity problems are avoided in KBSA.

4.1. Requirements vs. Application Goals

Some high-level statements should be satisfied to the letter in the implemented system. For example, no aircraft should be controlled by more than one controller at once. We use the term *requirement* to refer specifically to these inviolable constraints on the system. Other constraints may not be satisfied exactly, or exceptional cases may arise where they are not satisfied at all; we call these *application goals*. One such application goal is to ensure that aircraft follow their flight plans. It may not always be possible for the ATC facility to cause aircraft to follow their flight plans. Nevertheless, stating the goal explicitly helps to clarify the purpose of the actions that the ATC system actually performs.

We make a distinction here between *application goals*, which are goals which the application must meet, and *development goals*, which describe actions that the specifier wishes to perform in refining the specification. Application goals describe desired behavior of the application; development goals describe individual tasks that the specifier must perform on the road to producing a low-level specification of the desired behavior. For example, the specifier may identify an application goal that the aircraft must never collide. Then during the specification development the specifier may have a development goal of refining the definition of the air traffic control system so as to minimize the number of collisions.

In KBSA, one typically represents both application goals and requirements as Gist invariant declarations. For example, one might state the following:

```
invariant for all a1,a2 aircraft |
```

```
aircraft-position(ac, expected-position(ac,?))
```

This says that every aircraft's expected position is the same as its actual position. `expected-position` would in turn be derived from the flight plan.[2]

High-level editing commands can be employed to transform the invariant into an executable form. One editing command along this road is the command Maintain-Invariant-Reactively, which constructs a demon to reassert the invariant should if ever fail to hold. It generates the following:

```
demon correct-position[ac | aircraft]
  when ~aircraft-position(ac, expected-position(ac, ?))
  do update? aircraft-position(ac, ?) to expected-position(ac, ?)
```

This demon states that whenever the aircraft's position is not the same as the expected position, update it to be the same as the expected position. Performing such a transformation on an application goal is called *compromising* the goal, because it moves away from complete satisfaction of the goal. Substituting this demon for the invariant is a goal compromise because now the aircraft can go off course; there may be a time delay between when the aircraft goes off course and when the demon corrects it.

Not only does KBSA provide the necessary editing commands, it also provides a check to ensure that application goals are kept distinct from requirements. By default, any invariant is considered a goal, and can be compromised. However, the user may mark an expression as a requirement, in which case meaning-changing high-level editing commands cannot be applied to the expression.

4.2. Perfect Knowledge

In high-level specifications, we specify systems without regard for what data is accessible to which agents in the system. All agents are presumed to have perfect knowledge about the actions and properties of all other agents. Goals and requirements can then be defined in the most concise manner possible. As high-level specifications

---

[2] In Gist syntax the symbol | means "is of type", || means "such that". The symbol ? always appears inside some relational expression; it refers to some object for which the relation holds. Thus `expected-position(ac,?)` is equivalent to the `position || expected-position(ac, position)`. See [4] for a description of Gist syntax.

are refined into low-level specifications, data-access considerations are introduced; the specification of each agent is reformulated so that its behavior is described only in terms of the information available to it.

For example, when a plane arrives in the airspace of an air-traffic control facility, the controller who is supposed to assume control of the aircraft should be notified. However, the air-traffic control system cannot directly observe aircraft positions; it can only observe radar data. These data may have errors, and furthermore it is impossible determine directly which aircraft corresponds to which radar track. The correspondence can only be inferred by examining the flight plans for aircraft that are expected to arrive.

An example of a high-level editing command which assists in the transformation of this goal is the command Splice, which is described in some detail elsewhere [14, 11]. This command replaces references to a relations that should not be observable, e.g., `aircraft-position`, with a new relation `track-position`, which is computed by the radar. Splice also adds an invariant constraining the track's position and the aircraft's position to be the same. As this invariant is weakened, the specification becomes more realistic (and more complex).

In order to motivate the use of Splice, one must make clear what data is accessible to which agents. We provide the specifier with an annotation language for describing which agents can access which data, e.g.,

```
relation aircraft-position inaccessible-by atc-system
```

The specifier can first write specifications assuming perfect knowledge, and then add annotations describing data accessibility. Analysis of how data is used in a specification can then identify agents which access data which should be inaccessible to them; the specifier may then employ commands such as Splice to correct the information flow.

4.3. Unlimited Capabilities

Just as it is convenient to describe agents as if they had perfect knowledge, it is convenient to describe describe them as if they had unlimited capabilities. That is, agents can affect the environment directly, without going through some intermediate agent. A good example here is keeping the aircraft on course. In the demon shown in Section 4.1, the aircraft's position is being directly updated. In the low-level specification, some procedure will have to be defined which will have the effect of causing the aircraft's position to change (e.g., send a course correction request to the pilot of the errant aircraft).

The notion of the air-traffic control system "updating" the aircraft's position may seem strange. In order for it to make sense, think of updating not as modifying some fact in a database, but as causing some fact to be true in the world. By ignoring at the high level what capabilities agents have, one can describe what events agents cause to happen, without describing how they cause them to happen. This ability to describe "what" instead of "how" is what distinguishes specifications from implementations. In this way low-level specifications can be regarded as implementations of high-level specifications [9].

In order to distinguish "updating" from "causing to happen", and in order to guide the refinement of the high-level specification, the specifier must indicate which data is modifiable by which agents. This is done using annotations, as was the case in describing accessibility of information.

The notion of describing effects regardless of whether or not an agent is capable of causing the effects extends beyond the issue of modifiability of data. It is sometimes convenient to describe an effect even if no agent in the system has a method for achieving the effect. In such cases the specifier may use the **achieve** operator, a high-level [illegible] applied to Gist definitions. This **achieve** operator is similar to the achieve operator of Dershowitz [8]. For example, we could use the **achieve** operator in the correct-position demon as follows:

```
demon correct-position[ac | aircraft]
```

```
when ~aircraft-position(ac, expected-position(ac, ?))
do achieve[] postcondition
     aircraft-position(ac, expected-position(ac, ?))
```

The presence of an achieve operator indicates that the specification is operationally incomplete; some action must be added which causes the stated postcondition to be true. Pure Gist is has fully operational semantics: if a predicate is to be true in a behavior, then the specifier must define some procedure or demon which is capable of making the predicate true. The achieve operator violates this condition. Thus each achieve operator ultimately leads to a development goal being posted to supply an implementation for the achieve operator.

achieve operators are sometimes introduced by high-level editing commands to indicate where details must be filled in. The command Maintain-Invariant-Reactively sometimes does this. If it is unable to determine how to write a demon to cause an invariant to be satisfied with the permitted capabilities, it will insert an achieve operator in the demon, as in the above example. The specifier can then fill in the details that the command did not know how to fill in.

5. Guiding the Refinement Steps

Most of our development effort so far has been directed toward construction of high-level editing commands to support our specification refinement methodology. However, high-level editing commands are only part of what one would need for highly automated development assistance. We would ultimately like KBSA to take a more active role in the development process. We can see what role high-level editing commands play by regarding specification development as traversing a problem space. Each state in this space is a partially completed specification. The specifier is attempting to reach some goal state, in which a specification which adequately meets the specifier's requirements has been constructed. High-level editing commands are the operators that move from one state to the next. KBSA's responsibility currently is to execute the operators that move the specification from problem state to problem state. The user must decide whether or not a goal state has been reached, and if not what operator to apply.

We see the process of deciding what high-level editing command to apply as follows.

- The user starts with some *expectations* of what the goal states should look like. That is, he is able to analyze or validate a specification and determine whether or not it is complete. These expectations may be general well-formedness conditions that any completed specification of any problem should meet. For example, in a completed specification all agents should have a well-defined set of input and output data paths. Alternatively, there may be problem-specific expectations: the specifier has a certain kind of system in mind, and will be satisfied what the specification matches his intentions.

- The expectations are used to identifies an *issue* in the specification that need to be resolved. That is, the specifier decides that some particular aspect of the specification needs refinement. For example, the fact that the definition of `monitor` referred to `readings` was an issue that needed to be resolved.

- Selection of an issue results in a *development goal* being posted; in other words, the specifier has the goal of resolving the issue. The above issue of the `monitor` referring to `readings` results in the goal of eliminating references to `readings` within `monitor`.

- Finally, an editing command is selected to achieve the development goal, based upon knowledge of what kinds of goals each editing command is capable of achieving.

We are currently undergoing efforts to allow KBSA to contribute to the decision process at each stage. First, we are developing classifications for the type of development goals that high-level editing commands achieve. These goals fit into general classes, such as "move towards implementation", as well as specific, such as "reroute a data path". If the user states his or her goal, the system can suggest applicable editing commands.

Evaluating expectations and identifying issues is both promising and problematic. Some kinds of expectations are easy to test, given the representation of specifications that we have been developing. For example, a complete specification should not contain any `achieve` operators; each `achieve` is a potential issue to be resolved. We are developing analysis tools for checking other kinds of expectations, some of which are described in [14]. Problem-specific expectations, however, require KBSA to have more

knowledge about the specification. This knowledge must ultimately come from the user. We therefore are faced with a tradeoff situation: when will a user find it advantageous to describe his expectations to the system, and when will he prefer to do the analysis himself? We are in the process of empirically evaluating this question. In the mean time, we are including representations in KBSA for a wider range of specification knowledge, such as what data is observable by whom, and whether or not high-level requirements can be compromised. We then build high-level editing commands that would be selected on the basis of this knowledge, and determine empirically which is the most satisfactory way of controlling the specification development.

6. Relation to Other Techniques

The method of constructing high-level specifications and refining them into low-level specifications bears some resemblance to informal methods of requirements elicitation (e.g., [7, 12]). As in these other methods, the specifier attempts to describe some aspects of a system as directly as possible, while ignoring other aspects of the system. Where we differ from other techniques is in the order in which we specify things. In DeMarco's technique, for example, one specifies data flow first, and does not start specifying the functionality of the system until after the data flow is mapped out. One never specifies the requirements of the system as a whole; instead, one specifies the requirements on each node in the data-flow graph. In our approach, one starts by describing the requirements and goals for the system as a whole, and then starts to define the data-flow organization.

The advantage that we see in the KBSA approach is that the specification is operational even at the high level. That is, it is possible to symbolically execute high-level specifications. The same is not true for data-flow diagrams. Psychological studies of software design [1, 15] point to the importance of "mental simulation" as a tool for analyzing partial designs. Using KBSA, a specifier does not have a simulate a specification in his or her head; KBSA can do the simulation automatically.

High-level specifications are particularly important in deciding how to introduce

computing systems into an organization. Using DeMarco's data-flow approach, for example, requirements analysis first involves sketching the data flow in the existing organization. Then when the time comes to describe how the new computing system will affect the organization, it is necessary to identify all of the data-flow nodes that might be affected, draw a circle around these nodes, erase everything inside the circle, and redesign the contents of the circle from scratch. Thus there is no assurance that the new system organization meets the same requirements that the old organization does. Using our approach, the high-level requirements and application goals carry over into the new system organization, simplifying the redesign task.

At the same time, we recognize that data-flow diagrams are a useful technique for doing requirements analysis. The Requirements Assistant provides support for data-flow diagrams, and rightly so. In the ideal case, presentation techniques such as data-flow diagrams and goal refinement capabilities would be integrated into a single system. At that point, the distinction between requirements analysis and specification development would break down. Both would be aspects of the same process, one of developing formal specifications from ill-defined requirements.

7. Conclusions and Future Research

KBSA's approach to specification development shows promise as a way of allowing specifiers to state their goals for a system as concisely as possible, and build them into a coherent specification. Already we find that the high-level editing commands that we are developing are used over and over again in a variety of contexts. We intend to continue to develop this library.

The next major test for this approach is to use it as a basis for explanation. We believe that specifications that have been developed from high-level specifications are easier to understand. In fact, it should be possible for KBSA to generate explanations for specifications automatically. The record of specification development will thus become more than a corporate memory; it will be come a corporate consultant, able to explain software systems to people who were not involved in developing the specifications. If we are successful in this regard, then the refinement-based approach

will shown to be than much more powerful.

8. Acknowledgements

The author wishes to thank the members of the Knowledge-Based Specification Assistant Project for their contributions to the work described here: Robert Balzer, Bill Swartout, Don Cohen, Martin Feather, Jay Myers, Kai Yue, Neil Campbell, Anno Langen, and Ed Ipser. We also wish to thank Lt. Kevin Benner of RADC, as well as John D'Addamio, David Harris, and others at Sanders Associates for providing us information about the air-traffic control problem.

This research is supported by the Air Force Command, Rome Air Development Center under Contract No. F30602-85-C-0221. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of RADC, the U.S. Government, or any person or agency connected with them.

References

1. Adelson, B., and Soloway, E. "The Role of Domain Experience in Software Design". *IEEE Trans. on Software Engineering SE-11*, 11 (November 1985).

2. Balzer, R., N. Goldman. "Principles of Good Software Specification and Their Implications for Specification Languages". Proceedings of the Specifications for Reliable Software Conference , Boston, Massachusetts, April, 1979, pp. 58-67. (Also presented at the National Computer Conference, 1981).

3. Balzer, R., C. Green, T. Cheatham. "Software Technology In The 1990's Using A New Paradigm". *Computer 18* (November 1983), 39-45.

4. Balzer, R. "A 15 Year Perspective on Automatic Programming". *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985), 1257-1268.

5. Cohen, D. Symbolic execution of the Gist specification language. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, IJCAI, 1983, pp. 17-20.

6. Cohen, D. A Forward Inference Engine to Aid in Understanding Specifications. Proceedings of the National Conference on Artificial Intelligence, AAAI, August, 1984, pp. 56-60. Also available as USC-ISI/RS-84-135.

7. DeMarco, T., *Structured Analysis and System Specification*. Yourdon Press, 1978.

8. Dershowitz, Nachum. *The Evolution of Programs*. Birkhauser, Boston, Mass., 1983.

9. Feather, M., P. London. Implementing Specification Freedoms. Tech. Rept. RR-83-100, ISI, 4676 Admiralty Way, Marina del Rey, CA 90291, 1983. (Also appeared in "Science of Computer Programming 2, 1982").

10. Feather, M.S. A Survey and Classification of Some Program Transformation Approaches and Techniques. Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Toelz, FRG, April, 1986, pp. 165-195.

11. Feather, M.S. Constructing Specifications by Combining Parallel Elaborations. To appear in IEEE TSE.

12. Finkelstein, A. Making Formal Specifications Dynamic Objects. Proceedings of the Software Process Workshop, 1987, pp. 133-136.

13. Goldman, N. and D. Wile. Gist Language Description. Draft.

14. Johnson, W.L. Overview of the Knowledge-Based Specification Assistant. Proceedings of the Second Knowledge-Based Software Assistant Conference, Rome Air Development Center, 1987.

15. Kant, E. "Understanding and Automating Algorithm Design". *IEEE Trans. on Software Engineering SE-11*, 11 (November 1985).

16. Manna, Z., and Waldinger, R. "Synthesis: Dreams => Programs". *IEEE Trans. on Software Engineering SE-5*, 4 (July 1979).

17. Sanders Associates. Knowledge-Based Requirements Assistant - Interim Technical Report. Software Systems Engineering Directorate, March, 1986.

18. Swartout, W. GIST English Generator. Proceedings of the National Conference on Artificial Intelligence , AAAI, 1982.

19. Swartout, W. The Gist Behavior Explainer. Proceedings of the National Conference on Artificial Intelligence, AAAI, Washington, D.C., 1983. (Also available as ISI/RR-83-3).

SOFTWARE DESIGN RESEARCH AT DARPA

SLIDES SHOWN

19 August 1987

W. Scherlis (DARPA/ISTO)

KBSA '87

Technology Focus

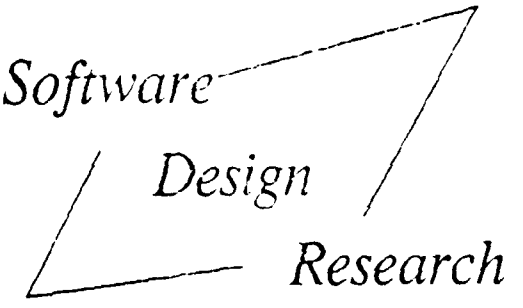
(1) Creation of a new paradigm for software systems **design** and life cycle **management**.

Transition to a view of software design information (in the form of specification and design components) as an **asset** that accumulates and that can be applied to many problems.

(2) Embodiment of research results in the form of **environments** for design and maintenance of systems.

Explicit **transition** approach involving the Ada, Lisp, and scientific communities.

PROGRAM GOALS



IMPROVE PRODUCTIVITY

- Increase reliabilty and performance
- Minimize cost and duration of process

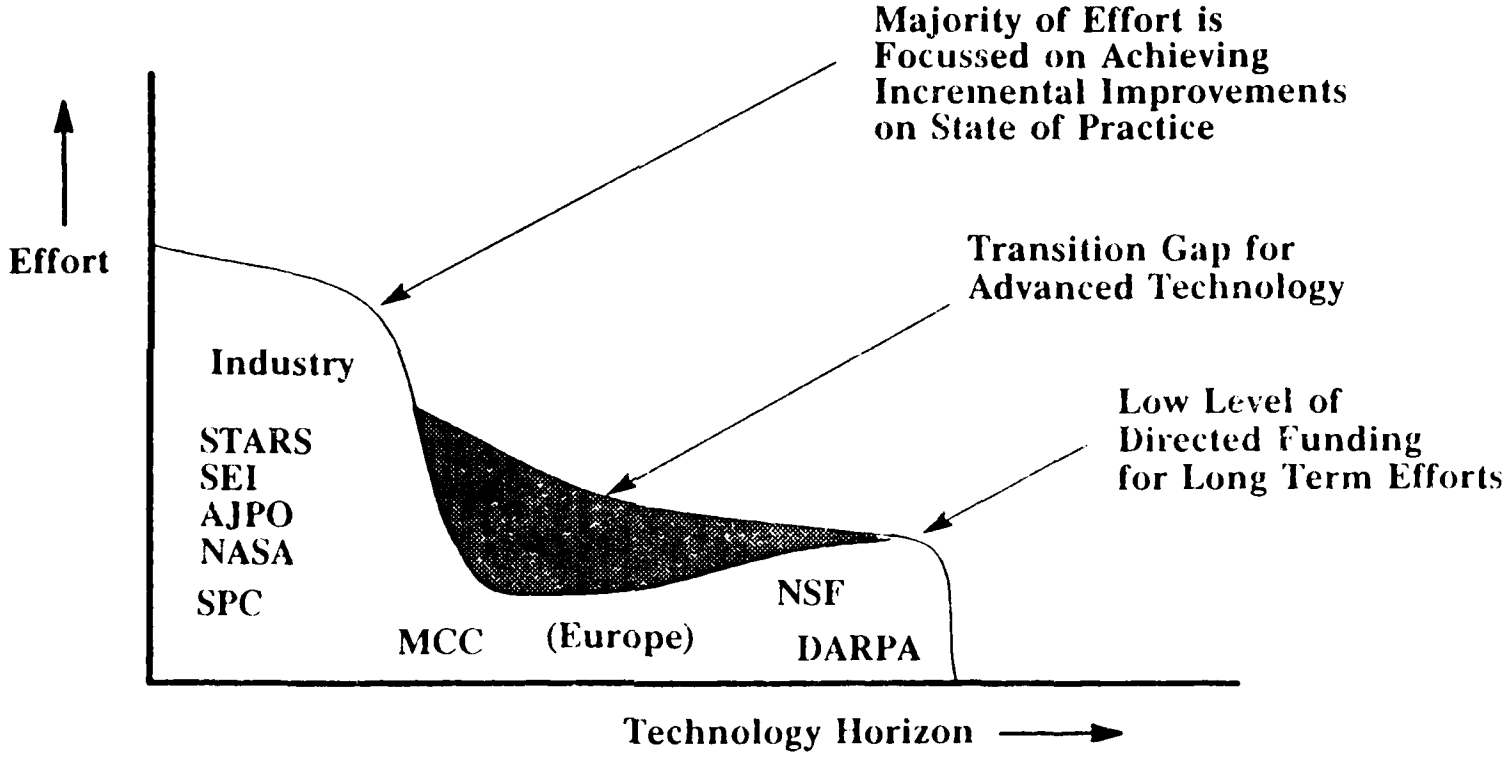
ACHIEVE NEW CAPABILITY

- Rapid adaptability
- Trust
- Real-time systems

TECHNICAL FOCAL POINTS

- Design retention and management
- Reuse and capitalization
- Exploitation of parallelism

US SOFTWARE RESEARCH FUNDING



(Qualitative rendering.)

• DESIGN: RETENTION AND MANAGEMENT

Objective: **Retain and apply design information gained during software development.**

Rationale: Programs describe machine behaviors, but do not contain enough design information to describe the *intent* of the behaviors.

Programmers, maintainers, and verifiers require this design knowledge and must now recover it at great expense.

Impact: **Direct use of design information will enable use of a new paradigm based on accumulation and application of design assets.**

• DESIGN: ADAPTATION AND REUSE

Objective: **Build tools to support the creation of systems whose design components are reusable.**

Rationale: The abstract nature of software creates a great potential for adaptability.

Current methodologies and languages nonetheless yield systems that are overcommitted and brittle.

Impact: **Software tools will enable acquisition and use of design information to rapidly adapt already deployed systems.**
Tools will assist in sustaining trust.

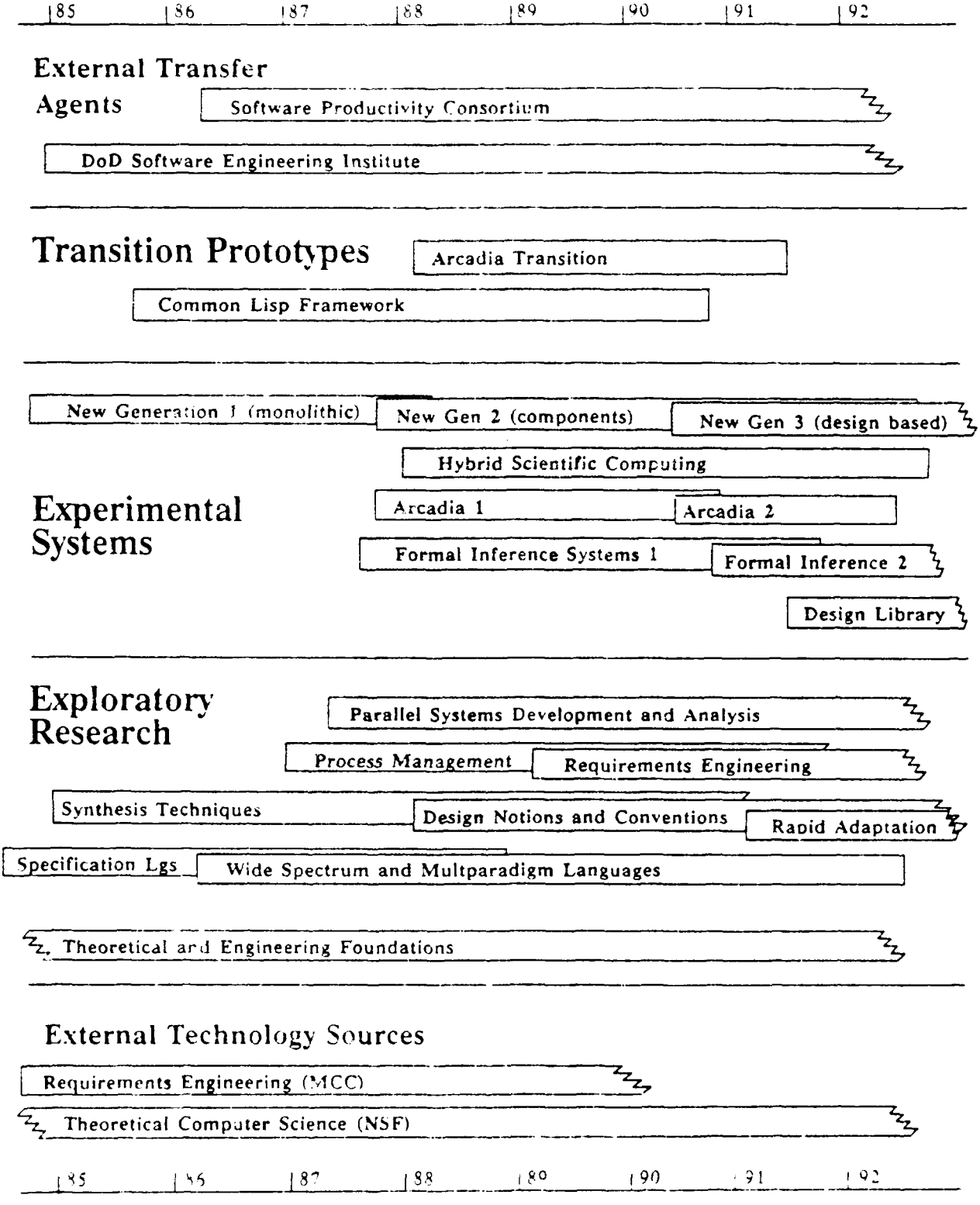
- **PARALLELISM**

Objective: **Create software technology to enable full exploitation of the potential power of the parallel computing architectures.**

Rationale: Current languages, algorithms and tools are oriented toward sequentiality.

Software technology progress is required for systems scalability, reliability, and full exploitation of parallel performance potential.

Impact: **Languages, algorithms, and tool support will provide open access to scalable architectures.**


85
86
87
88
89
90
91
92
External Transfer Agents
Software Productivity Consortium
DoD Software Engineering Institute
Transition Prototypes
Arcadia Transition
Common Lisp Framework
New Generation 1 (monolithic)
New Gen 2 (components)
New Gen 3 (design based)
Hybrid Scientific Computing
Experimental Systems
Arcadia 1
Arcadia 2
Formal Inference Systems 1
Formal Inference 2
Design Library
Exploratory Research
Parallel Systems Development and Analysis
Process Management
Requirements Engineering
Synthesis Techniques
Design Notions and Conventions
Rapid Adaptation
Specification Lgs
Wide Spectrum and Multparadigm Languages
Theoretical and Engineering Foundations
External Technology Sources
Requirements Engineering (MCC)
Theoretical Computer Science (NSF)
85
86
87
88
89
90
91
92

Management and Transition Strategy

(1) **Quality.** Use of the best research groups and best resources. Creation of new organizational structure in the community to facilitate creation, sharing, and transition for a new culture.

(2) **Scarcity** of high quality groups requires explicit management and transition to new approaches.

(3) Explicit attention to **engineering** in support of the research activity and in support of transfer.
Top–down vision, bottom–up realization.
No monoliths.

(4) Development and sharing of **component technologies**, including **subsystems**, **languages**, and **conventions** for abstract interfaces.
Transition of current prototypes into enabling technologies for next generation.

Subsystems: Mach, X, [Syntax], [OOdb], ...

Languages: Ada, Lisp, WSLs, SpecLgs, ScientificLgs, ...

Conventions: OOdb access, Unix files, ...

(5) Confluence of three major participating **communities.**

(6) **Transfer** plan with Ada community, SEI, Lisp.

(7) Explicit attention to **foundational** work in support of programmatic goals.

Plans and Meta-plans
in an
Intelligent Assistant
for the Process of Programming©

Karen E. Huff
Victor R. Lesser

Computer and Information Science Department
University of Massachusetts
Amherst, MA. 01003

May, 1987

Abstract: We describe an architecture for providing intelligent assistance to the programmer carrying out the *process* of programming. This architecture, based on an AI *planning* paradigm, can provide both passive and active assistance. Passive assistance, accomplished by *plan recognition*, is used to detect and avert process errors. Active assistance, accomplished by *planning*, is used to automate the programming process. A key issue in achieving appropriate levels of assistance is the ability to capture complex domain knowledge in a planning formalism. We illustrate several problems in traditional hierarchical formalisms, and propose a solution based on the use of meta-plans that dynamically transform plans.

1.0 Introduction

Development environments of today provide little assistance to the programmer with the *process* of programming. The tasks of mapping programming goals into sequences of tool invocations, revising plans when results are not as expected, and performing the required — but mundane — housekeeping chores such as file management are carried out with only rudimentary support. As new techniques are developed to automate some part of the process, a new tool is created (or an existing tool expanded). This leads to a situation where, by definition, the process of programming comprises all the activities that cannot be fully automated.

This inability to provide full automation does not, however, preclude other forms of assistance. Two possibilities, based upon reasoning about the programming process, appear promising. In or case, the programmer retains the initiative for performing the process, issuing commands exactly as at present. A passive intelligent assistant *monitors* these actions, measuring them against its (extensive but still incomplete) knowledge of the process. In this mode, many types of process errors could be detected and averted. Such an assistant would be an automated version of a colleague watching over the shoulder of a programmer at work. In the second case, the programmer relinquishes control to the intelligent assistant, specifying only goals to be achieved rather than the detailed commands by which they are to be achieved. Here, an active intelligent assistant *plans* a sequence of commands using its knowledge of process actions; since its knowledge is incomplete, the programmer must supply certain decisions which are beyond the scope of the assistant. In this mode, a cooperative automation of the process is achieved.

1.1 Architecture for Intelligent Assistance

We are developing an architecture (diagrammed in Figure 1) that provides both of these forms of assistance in order to offer the programmer a very powerful and flexible support environment. Our approach is based on the use of AI planning techniques, which offer a well-developed framework for reasoning about actions. Planning technology represents one possible route towards "process programming" [14], a concept that is the subject of current debate [10]. The plan-based architecture, named GRAPPLE , that we are currently developing [2] builds upon earlier work in intelligent assistant architectures [1,3,5,6].

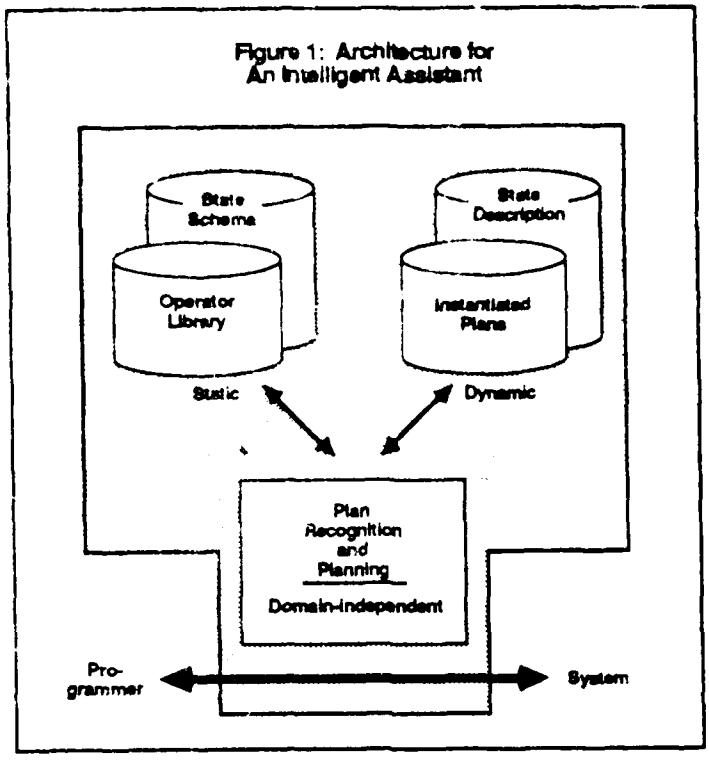


Figure 1: Architecture for An Intelligent Assistant

Under a planning paradigm, knowledge of the process is expressed as *operators* defining the legal actions of programming, together with a *state schema* defining the predicates that describe the state of the programming world. A *plan* is a partial order of operators (with all variables bound) that achieves a goal given an initial state of the world. The algorithms for monitoring programmer actions are the algorithms of *plan recognition*, where a plan to achieve some goal is identified incrementally from sequences of actions performed by the programmer. The algorithms for carrying out a programmer goal are the algorithms of *planning*, where a partial order of operators is generated to achieve the stated goal.

A major benefit of this approach is that the intelligent assistant is domain-independent. Merely by changing the library of operators and associated state schema, alternative programming processes or different toolsets may be accommodated; tailoring the assistant to project-specific policies is similarly accomplished. Enlarging the library of operators allows coverage of additional life-cycle phases (and the all-important feedback loops among phases). The intelligent assistant can act at the operating system command level and/or within a complex tool (by considering the functions provided by the tool to be tools themselves.)

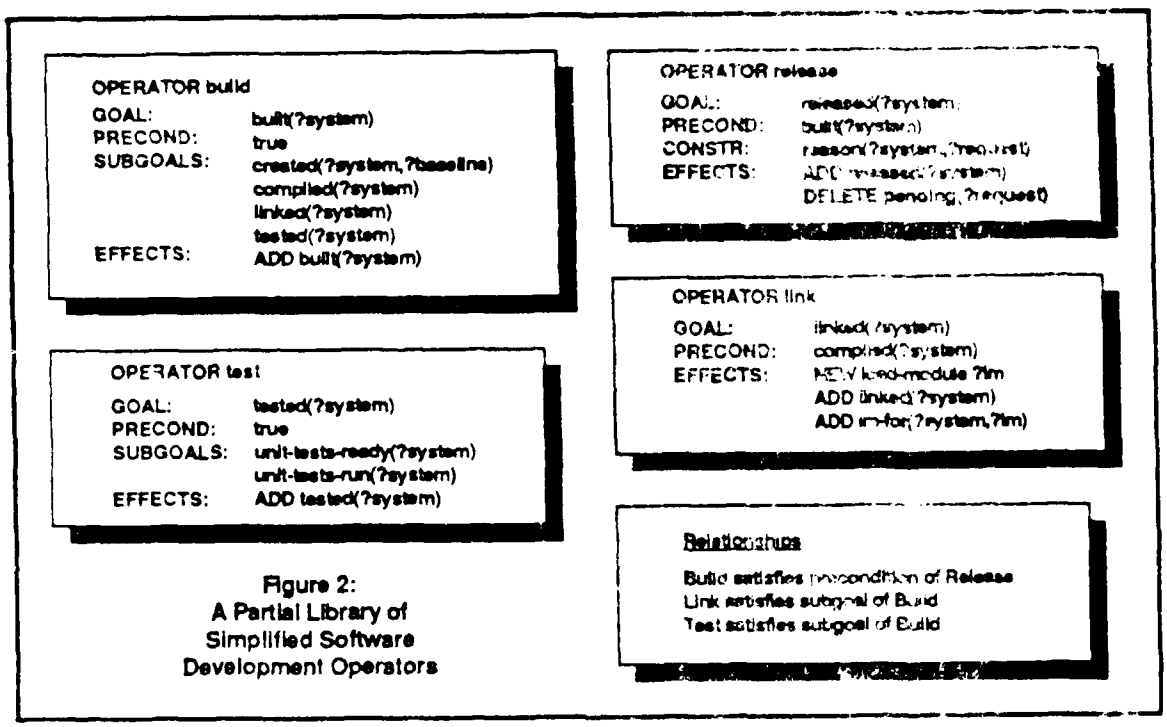


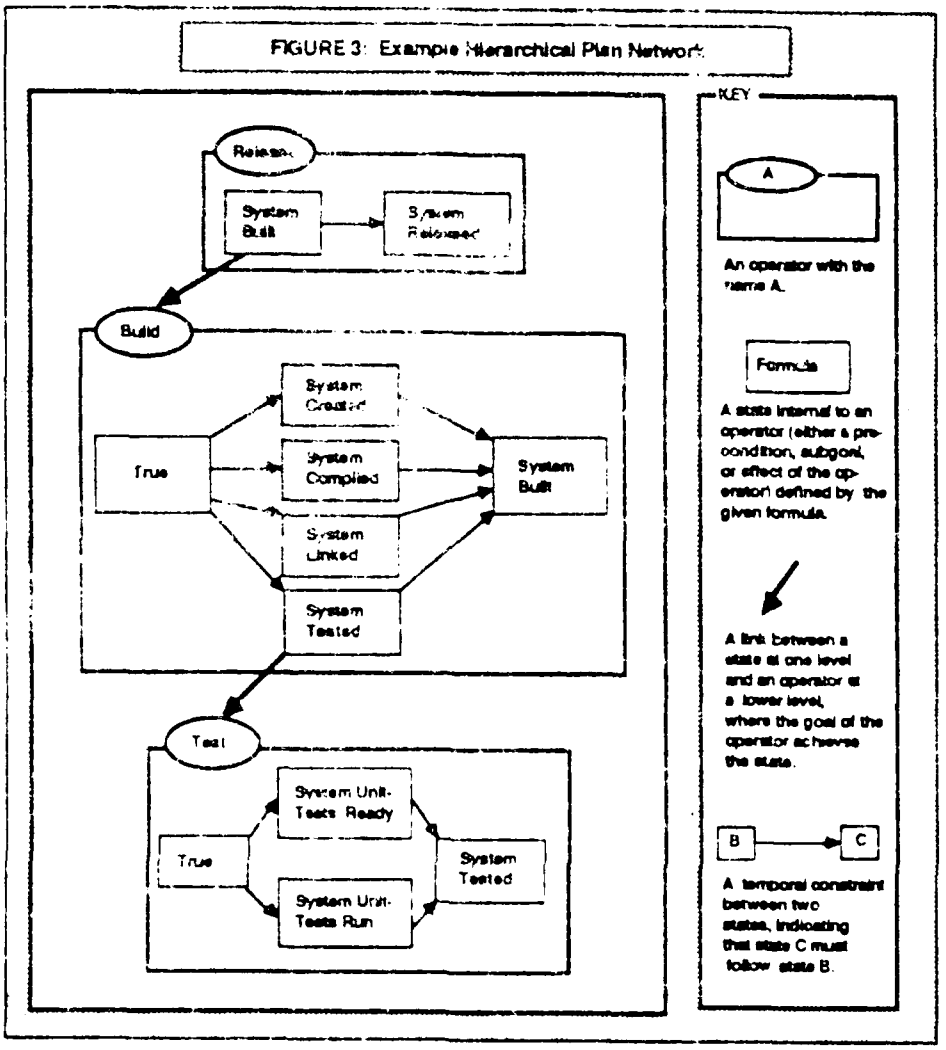
Figure 2:
A Partial Library of Simplified Software Development Operators

1.2 Processes as Plans

As a test domain for this intelligent assistant, we are exploring the programming process as it is carried out for a traditional programming language such as C at the command level in an existing operating system, UNIX™, assuming a style that follows accepted engineering practice. A partial library of operators for this domain is sketched in Figure 2; these operators have been greatly simplified for purposes of this example, but should serve to indicate the general nature of the approach.

The operator definitions follow standard state-based, hierarchical planning approaches [15,17,19]. In a state-based approach, each operator has a *precondition* defining the state that must hold in order for the action to be legal, and a set of *effects* that defines the state changes that result from performing the action. These core clauses are augmented by a *goal* clause that defines the principal effects of an action (thus distinguishing them from "side-effects" of the action), and a *constraints* clause that defines restrictions on parameter values. A hierarchical approach supports multiple levels of abstraction in operators by allowing the definition of complex operators having *subgoals* that decompose the operator into simpler

UNIX is a registered trademark of AT&T Bell Laboratories.



subproblems. Primitive operators, which do not have subgoals, correspond to the atomic actions in the domain.

The basic data structure of a planner or plan recognizer is a hierarchical plan network as first developed in [15]. An example of such a network (using some of the operators of Figure 2) is given in Figure 3. There, a vertical slice through a network covering three hierarchical levels is shown, with the highest level at the top of the figure. Downward arrows between levels connect desired states with operators instantiated to achieve them. Such instantiated operators consist of additional states describing the internals of the operator: preconditions, subgoals, and effects. Arrows within levels show how the achievement of certain states is partially ordered with respect to time. Some orderings are dictated by the operator definitions: precondition states must always precede subgoals, for

example. Other orderings are imposed to resolve interactions between operators that could destroy the plan. Orderings are propagated from level to level, but have been omitted to simplify the figure.

Both planning and plan recognition involve building a complete plan network. This is done by actions such as choosing operators to achieve states, instantiating these operators, and resolving conflicts between newly revealed states and existing states. The strength of a planning approach lies is this ability to handle conflicts that would otherwise destroy a plan. For example, consider a situation where an operator has a two part precondition, requiring that both A and B be true, and the only operator available for achieving B also achieves NOT A. Any plan that allowed the operator for achieving B to follow the operator for achieving A would fail, due to this interaction of the operators. A viable plan must require that the operator to achieve B precede the operator to achieve A. While re-ordering operators solves a common type of interaction, other means of conflict resolution have also been developed [15,17,19].

Planning techniques are needed when the chosen problem representation has rules that are not decomposable [13], i.e., when solutions to parallel subproblems cannot be tackled independently and trivially recombined. Certain types of production rules (and expert systems based on them) assume decomposability. While such rule systems have been used to handle some types of process automation for programmers [9] the approach is not readily extensible, for example, the precondition interaction described above cannot be handled. Additional interactions are certain to arise when multiple plans are simultaneously in progress, as is often the case with programming work. For example, when a programmer is both fixing a bug in an old version of a system and adding new functionality in the latest version, different directories must be used to separate the two sets of source and object modules.

1.3 Assistance from a Planning Perspective

Together, planning and plan recognition make it possible to deliver a broad range of services to the programmer. The planning perspective suggests ways that the specific services can be accomplished:

- **Agendas and Summaries:**
  An agenda is the set of goals yet to be satisfied in a plan, and a summary is the set of goals that have been satisfied. Either can be described at various levels of abstraction, given the hierarchical nature of the plans. Both are "intelligent" in that the system has the knowledge to interpret what they mean: for example, a plan for carrying out an agenda item can be constructed.

- **Error Detection:**
  Three different classes of errors can be handled. Logistical errors represent faulty planning by the programmer. Examples are executing an action before its precondition is met, or undoing a previously satisfied precondition before the relevant action has started. Housekeeping errors represent errors of omission. Examples are keeping files in the "right" directories, checking sources back into a source code control system, deleting extraneous files, and perhaps adhering to certain project-specific policies. The final class of errors are substantive software process errors: for example, violating constraints in operators such as making the wrong choice of a baseline from which to develop a new system.

- **Error Correction:**
  The error correction facilities amount to an "intelligent" do-what-I-mean. Types of corrections (related to the types of errors described above) include re-ordering actions, identifying missing activities, supplying a plan to satisfy a required state and substituting bindings that satisfy required constraints.

- **Query Support:**
  Queries as to either the state of the world or the state of the actions can be handled, since both states are maintained to support planning functions. Each state represents a "database" of information about current status and past history.

- **Cooperative Automation:**
  The automation itself is achieved by planning. Cooperation is accomplished by requesting programmer decisions on such issues as what parameter bindings to use in operators, when to terminate iterated activities, or how to select among alternative operators.

1.4 Achieving Intelligent Assistance

The architecture we have described is an ambitious one. While planning appears to be the right framework for reasoning about a process, the key is being able to capture all the relevant process knowledge in the operator definitions. If too little knowledge is captured, the intelligent assistant will be rigid, and ultimately more constrictive than supportive. The challenge arises because the programming process is at least as complex as any domain previoulsy tackled for a planning application. And certain aspects, such as the inherent "trial and error" nature of programming and the fact that the intelligent assistant is not intended to be fully autonomous, are novel.

As a result of designing an operator definition language engineered to handle real-world domains [7] and writing software process operators in this language [8], we have encountered limits to the representational adequacy of existing hierarchical plan formalisms. There are problems in capturing such relevant domain knowledge as how to recover when an operator fails or when to use special case operators. The challenge is to provide this knowledge in a way that is tractable both to the planning algorithms and to the writer of the domain operators.

In the remainder of this paper, we illustrate the problems of representational adequacy with examples of software processes. We introduce a solution based upon dynamically transforming plans, as opposed to defining additional static operator definitions. We discuss how the transformations are formalized and expressed as meta-level operators; both the state description and required operator constructs are covered. Finally, we present status and conclusions.

2.0 Representational Power

Hierarchical plan systems, based on NOAH [15] and NONLIN [17], have strengths both in their planning algorithms and in the nature of their operator definitions. When describing a complex domain, the hierarchical approach is appealing because activities can be defined at different levels of abstraction, with more or less detail as appropriate. Another strength lies in the modularity of operator libraries: operators can be written without knowledge of the other operators in the library, and operators are potentially applicable in any context where their goals match the preconditions or subgoals of other operators. In complex domains, cases arise where appropriate expressive power is lacking

in the operator formalism, attempts to describe certain operators accurately can jeopardize the library modularity or fail outright. Consider the following problems.

2.1 Limits on Representational Power

Adding special case operators to a library may require that preconditions or subgoals of existing operators be rewritten. For example, when testing a system that is intended to fix certain bugs, the programmer should run the official testcases associated with those bugs, in addition to those testcases that would otherwise be selected. One solution is to write a separate operator covering all testing needed when bugs are being fixed; its precondition restricts its applicability to systems intended to fix bugs. Now there are two operators for testing that are intended to be mutually exclusive. Therefore, the normal operator for testing must specify in its precondition that it is not applicable to cases where bugs are being fixed[1].

In other situations, existing operators must be rewritten in artificial ways. Consider testing a system that is about to be released to a customer, such testing should include running the testcases in the regression test suite[2] (again, in additional to normally selected testcases). The precondition for this special operator concerns the existence of a goal to release the system; while the goal formula is expressible using domain predicates, the fact that a goal with this formula is currently instantiated is not expressible in domain terms. The only recourse is to write separate operators with artificially different goals. Then, operators (like *build*) that have testing subgoals will be affected. Thus, the designer of the operators must produce not only a normal *test* operator and a *test-for-release* operator, but also a normal *build* operator and a *build for-release* operator, to ensure that the right type of testing is performed in all cases.

Expressing special cases with this brute force approach, already attended with disadvantages, breaks down entirely when multiple special cases affect a single operator; the combinations are intolerable from the designer's perspective. Special cases are not

---

[1] One could institute a fixed preference strategy to select the operator with the most specific precondition that can be satisfied. However, in general this is overly restrictive — it would prevent a car buyer from financing a car purchase by selling stock to raise funds because taking out a car loan is the most narrowly applicable operator.

[2] In software engineering, regression testing is performed to ensure that bugs have not been introduced into functions that were previously shown to work correctly.

guaranteed to be simply additive with respect to the [illegible] operators must be provided for all *combinations* of [illegible]

In dealing with recovery from operator failure, there [illegible] selecting the right recovery operator with a failure situation, and [illegible] recovery strategy itself. Sometimes special operators are [illegible] and only for failure recovery. For example, one of the actions for [illegible] failure due to bugs in the compiler is to report the compiler bug. [illegible] written as an operator, but how will such an operator get [illegible] the constructs indicating what goals (and therefore what operators) should be [illegible] when a failure occurs. At other times, the recovery strategy may [illegible] operator in a special way. If the *build* operator fails because the [illegible] (as would be the case if the linker detected [illegible]), the strategy is to restart the build process using the [illegible] option to edit. Implementation of such a strategy requires access to [illegible] instances; [illegible]

2.2. Increasing Representational Power

These problems have previously been tackled [illegible] special operator-specific constructs covering selected cases [illegible] dependent strategies that can be tailored in fixed ways. Not [illegible] approach to the problem of creating special case operators is [illegible]. These policies derive their power from the fact that the [illegible] user to use plan-writing constructs such as *<policy> IMPLIES* [illegible] or *<policy> APPLIES TO <operator>*, in addition [illegible]. Recovery from failed operators is addressed in SIPE [illegible] special error recovery strategies, domain-dependent strategies, such as reactivating a goal (RETRY in SIPE) [illegible] parameterized to allow deployment in operator-specific ways.

We propose a single formalism that extends the representational power of hierarchical planning paradigms. Our approach is based on the observation that some domain knowledge is most naturally expressed in meta-level constructs dealing with a plan and its internal structure. Knowledge of special cases is not embodied in additional domain operators, but rather in a set of transformations by which plans are dynamically modified [illegible] ongoing set of special conditions (including failures). These [illegible] are specific to a given domain operator; rather, are meta-plans

composed from meta-level operators. These operators have the same form as domain operators (with preconditions, subgoals, etc.), but they use predicates describing plan networks, as well as domain predicates.

The primary advantage of this method is expressive completeness. Any aspect of an operator definition (such as preconditions, subgoals, constraints, or effects), as well as any aspect of an operator instance (such as bindings of variables or ancestor operator instances) can be accessed or modified. The transformational approach also addresses the problem of providing a complete library of operators. Because knowledge of exceptions is partitioned from knowledge of normal cases, the two issues can be tackled separately. The process of writing operators is further improved because multiple transformations can apply to a single operator, thereby preventing combinatorial explosion in numbers of operators.

3.0 Transformations on Plans

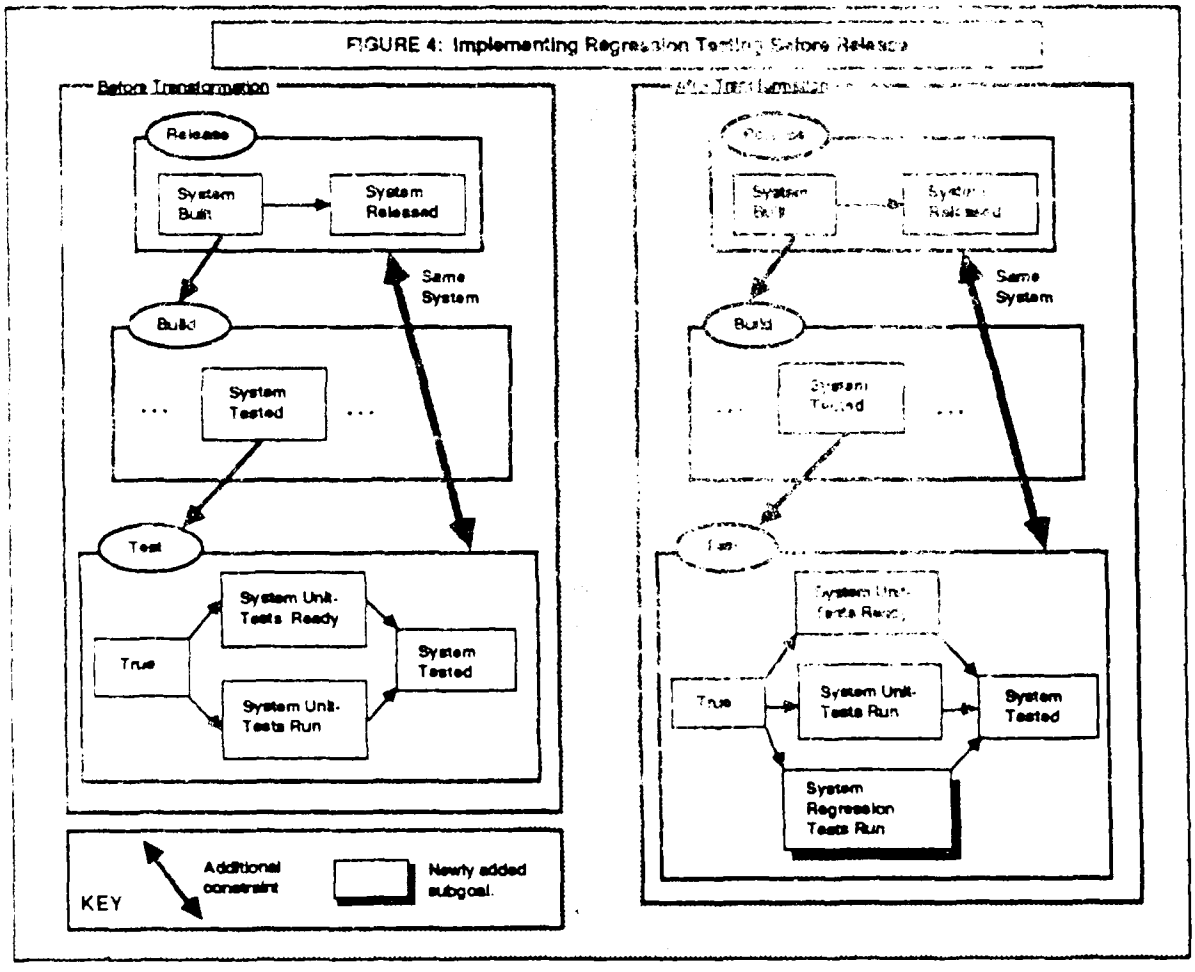
Both planning and plan recognition involve building a complete plan network. Normally each action applied to a network can be thought of as taking the network one step closer to completeness. In contrast, a transformation will reformulate the current state of the network, with the effect of changing the solution set that will be pursued to complete the network. Such a reformulation is necessary either because the existing state of the network does not accurately reflect special circumstances or because other actions have reached a dead end (for example, an operator in the plan has failed.) Reformulating the network represents a permanently- instantiated objective of the planner/recognizer. Thus, the execution of (top-level) transformations will be data-driven: they will be applied whenever the current state of the network indicates an opportunity to do so.

3.1 Example: A Special Case

Software development is an example of a domain where a large part of the knowledge about how actions are carried out is concerned with special cases. Consider a transformation that implements the requirement to do regression testing before a release. When expressed precisely, the transformation affects an instance of *test* occurring as part of the expansion of *release*. To be entirely safe, one additional restriction should be given: that the *system* being tested is the same as the *system* being released. This will allow other testing instances to occur in the same expansion (such as running a testcase to help decide what editing changes are needed), while ensuring that regression testing is required on the

right one. Expressing this condition requires access to the dynamic correspondence between the variable names used in the two operators. The BEFORE case of Figure 4 shows one situation in which this transformation is applicable.

Assuming the *test* operator of Figure 2, the effect of the transformation is to add an additional subgoal to run the regression test cases. The formula defining the new subgoal is supplied explicitly in the transformation -- it need not have appeared previously in the plan network. Only the one operator instance is modified; the basic operator definition for *test* is unchanged. The results of applying this transformation are shown as the AFTER case in Figure 4.



3.2 Example: Failure Recovery

Software development is a domain where there are many causes of failure, including system problems, tool problems and programmer error. In particular, given that much work is carried out on a trial-and-error basis, failures due to programmer error are to be expected frequently. Consider the case of the *link* operator failing to produce a load module because errors made by the programmer were detected. In fact, decisions about recovery from this failure are not made at the local level of the *link* operator; a *link* operator failure implies that the parent operator has failed, and the appropriate recovery strategy is dictated by what that parent operator is.

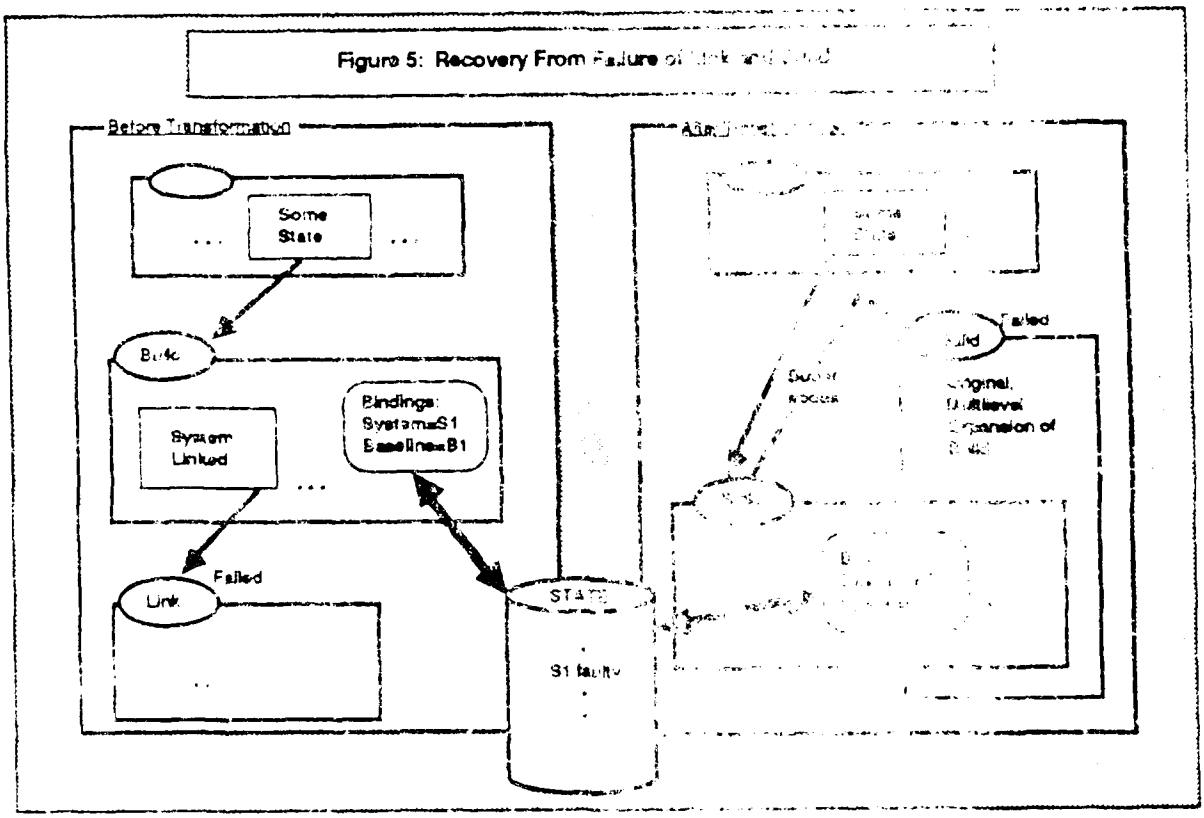
The parent operator will usually be the *build* operator. If the *build* operator has failed and the *system* that was built is faulty, one recovery scenario is to go through the whole *build* process again but, instead of starting from the same *baseline* used in the original *build* instance, this new *build* instance will start with the faulty *system* as the *baseline*. That is, the new *build* instantiation will use as the binding of its *baseline* variable the binding of the *system* variable from the failed *build* instantiation.

These strategies can be expressed in two separate transformations. The first transformation applies to instances of *link* that have failed; its effect is simply to mark the parent operator instance as failed. The second transformation applies to instances of *build* that have failed; its effect is to create a new instantiation of the *build* operator, and to fix the binding of the *baseline* variable in that instantiation to be equal to the binding of the *system* variable from the "superseded" instantiation of *build*. This is shown in Figure 5.

3.3 Other Examples

The software domain is particularly rich in opportunities for expressing domain knowledge in transformations. Some additional examples are:

- In a multi-user system, when the number of users logged-in is below a certain threshold, then commands will be submitted for foreground execution rather than to a background queue. One transformation, applying to all operators utilizing the background queue, can be written in lieu of an additional version of each such operator.



- Recovering from failed operators includes the cause of leaving extraneous files. One transformation could identify certain files created by operators in the expansion of a failed operator and instantiate goals to delete them. This transformation applies one change (instantiating goals for specific variable bindings) many times (for each selected file).

- The conservative editing style of frequently saving a snapshot of the file being edited also involves eventually deleting the intermediate snapshots. This too is a complex transformation, because the deletions cannot be specified until after the identity of the satisfactory version is known.

- Programmers generally follow a set of rules about how files are allocated to directories. However, in the heat of activity, a file may be created in the "wrong" directory. A transformation could trigger on this and instantiate a goal to move the file to the proper directory. Such transformations reestablish desirable domain states, in the manner of McDermott's primitive policies [12].

- An operation copying one file to another is used expressly for the purpose of preventing conflicts between two subsequent actions, one of which will

modify the file and the other of which will use the original form of the file. Copy can be written as a normal domain operator, but as in the case of operator failure, some connection still remains to be made between the goal of copy and a situation when it is appropriate to instantiate that goal. A transformation can be written to do this; the precondition for the transformation is that an adverse interaction between two planned actions has been detected. Here a transformation is used to augment the domain-independent forms of untangling operator interactions by defining a domain-specific strategy.

- A well-established strategy for dealing with the compiler having blown up when directed to compile at its highest optimization level is to try again with optimization turned off. If this results in a successful compilation, the programmer will settle for a load module which is only partially optimized, even if performance testing was planned. This transformation should both rephrase the goal to lower the optimization required and instantiate a goal to repeat the performance testing when a fully optimized load module can be produced. This is an instance of McDermott's notion of rephrasing a goal when no plan can be constructed to achieve it.

- If editing a source module consists of cosmetic changes only, then an alternative to recompilation is simply to acquire (and place in the appropriate directory) the object module of the previous version (assuming no include modules were also changed). However, it is bad practice to do this on a final release to a customer. Only by expressing this in a transformation can we ensure that good practice is followed. In this case the goal is rephrased to take advantage of special circumstances.

4.0 Formalizing the Transformations

It should be clear from the foregoing examples that the transformations are operators on the state of planning; these meta-level operators are combined to form meta-plans. Procedurally-implemented meta-plans were introduced by Stefik [16] to pursue control issues in planning. Declarative meta-plans were defined by Wilensky[18] in order to share knowledge between a planner and a plan recognizer. Neither of these meta-plan systems was used to modify operator instances by adding new subgoals, changing constraints and so forth. Meta-plans that could modify steps or change parameter bindings were defined for a natural language dialog understanding system[11]. In these meta-plans, the modifications were meta-plan parameters which were bound from information in the utterances.

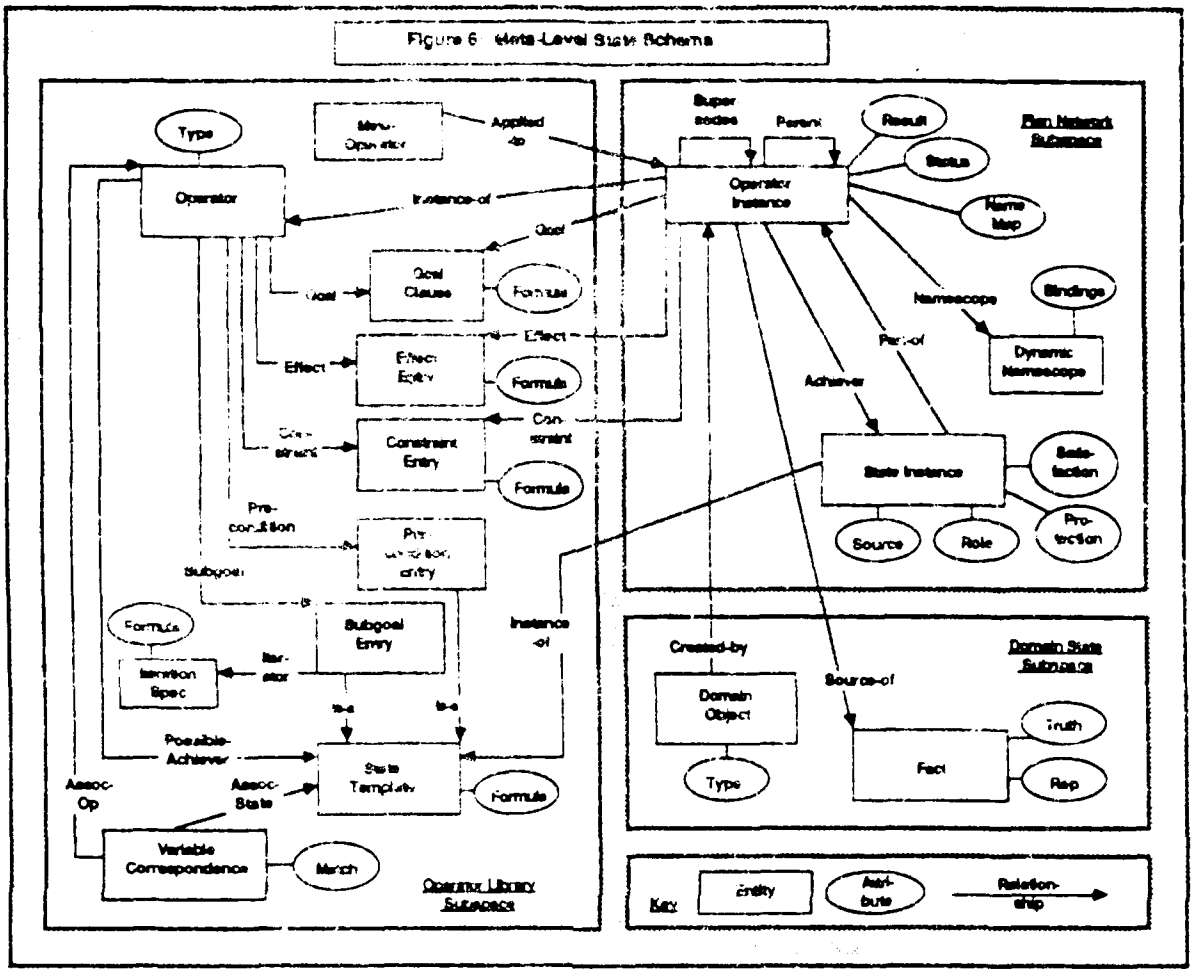
4.1 The Meta-level State Schema

The meta-level state schema covers most of the internal data structures used in planning. The entity-relationship (ER) model of data [4] provides a useful way to visualize such a complex state schema. In the ER model, there are entities which have attributes and participate in relationships with other entities. There is a straightforward translation between the ER model and predicate calculus, whereby relationships and attributes correspond to predicates. Semantic constraints can be expressed as axioms in predicate calculus. Other axioms can be used to define additional predicates and to define appropriate functions.

The ER diagram of the meta-level state schema is given in Figure 5. The objects and relationships shown are representative, not exhaustive. The schema describes operators as they are defined in the GRAPPLE formalism, and addresses the requirements of plan recognition (the context in which we are currently implementing the meta-plans).

The state schema for the meta-plans contains objects and predicates organized into three subspaces: operator library, plan network, and the domain state. The operator library subspace describes all components (preconditions, effects, etc.) of all operators, and their formulas and (static) variable names. The plan network subspace describes the hierarchy of operator instances: their dynamic status (started, completed, failed...), their internal states and status (pending, achieved, protected ..). Also associated with operator instances are the dynamic name scopes and variable binding information needed to evaluate operator formulas. The domain state represents the truth value of all domain predicates. Additional predicates cross subspace boundaries, relating operator instances back to their definitions, and operator instances to the domain predicates they affected.

The state schema is designed so that it represents a single choice from among the competing interpretations of a series of actions. Thus, if an operator can achieve the subgoals of two other operators, this will be represented using two states, each in a separate context. During recognition, there will usually be multiple contexts that are active; an important component of the recognizer comprises strategies for focusing among these alternatives, as well as for selecting the operations applied to an alternative. Thus we make the same distinction as in [16] between operations in planning space and the control strategies by which those operations are selected. The transformational meta-operators add to the number of operators subject to control decisions.



4.2 Meta-operator Constructs

Because the transformations are complex, expressing them as operators requires a language engineered for "real-world" use. For example, effects of operators must be able to create new objects (for example, a new subgoal instance). Some transformations (such as the one to delete extraneous files) require a facility for iterating subgoals: that is, for repeatedly achieving a subgoal formula over a set of bindings. Conveniently, all the needed facilities were already available in the formalism we designed to handle the complex domains anticipated for GRAPPLE [7]. No new facilities (except a keyword to distinguish between operators and meta-operators) were needed.

The transformation for regression testing before release is shown as a meta-operator (expressed in GRAPPLE notation) in Figure 7. This will be a top-level operator assuming

Figure 7:

```
(METAOPERATOR     regressions-before-release

(GOAL             applied-to(regressions-before-release,?test-op))

(PRECOND          (STATIC instance-of(?test-op,test) AND
                  ancestor(?test-op,?rel-op) AND
                  instance-of(?rel-op,release) AND
                  same-dynamic-name(system,?rel-op.
                     system,?test-op)) AND
                  NOT applied-to(regressions-before-release,
                     ?test-op) ) )

(EFFECTS          (NEW state-instance ?regressions)
                  (NEW subgoal-entry ?regr-subgoal)
                  (NEW iteration-spec ?iterate-info)
                  (ADD part-of(?regressions, ?test-op))
                  (ADD instance-of(?regressions, ?regr-subgoal))
                  (ADD iterator(?regr-subgoal, ?iterate-info))
                  (ADD source(?regressions, metaplan))
                  (ADD role(?regressions, subgoal))
                  (ADD protection(?regressions, not-protected))
                  (ADD satisfaction(?regressions, unknown))
                  (ADD formula(?regr-subgoal,
                        tested-on(?system,?regr-case)) )
                  (ADD formula(?iterate-info,
                        in-regression-suite(?regr-case)) )
                  (ADD applied-to(regressions-before-release,
                     ?test-op)) ) )
```

that its goal does not match the precondition of subgoal of other meta-operators; therefore, it will be executed when its precondition becomes true. That will happen when there is an instance of *test* whose ancestor is an instance of *release* AND when the *system* variables of both operators correspond (e.g., are mapped to the same dynamic name). The precondition carries the keyword STATIC, meaning that no explicit attempt should be made to render it true.

Performing the transformation is simply a matter of realizing the effects of the meta-operator in this case, because there are no subgoals to be achieved. These effects are all directed at creating a new state instance as part of the *test* operator instance. The new state instance has a supporting subgoal entry that defines the state formula and, since it is an iterated subgoal, the iteration formula. Note that the meta-operator contains these formulas explicitly; they consist of domain predicates and variable names in the static name scope of the *test* operator definition. After the transformation has been executed, the precondition

Figure 8:

```
(METAOPERATOR     propagate-link-failure

 :GOAL            status(?link-op,failure-processed) )

 (PRECOND         (STATIC status(?link-op, failed) AND
                  instance-of(?link-op,link) AND
                  query(?link-op, faulty(?system) ) ))

 (CONSTRAINTS     (parent ?link-op,?parent-op))

 (EFFECTS         (DELETE status(?parent-op,in-progress) )
                  (DELETE status (?link-op,failed) )
                  (ADD status(?parent-op,failed))
                  (ADD status(?link-op,failure-processed)) ))
```

will no longer be true for this instance of *test*; thus, this transformation is not meant to be applied more than once to the same situation.

The transformation exporting the failure of the *link* operator to its parent operator is shown in Figure 8, to show how a goal for failure recovery is expressed.

5.0 Status and Conclusion

A plan-based approach to intelligent assistance for the programming process appears to be an appropriate match between problem and solution paradigm. However, it is dependent on capturing complex domain knowledge in operator definitions. Transformational meta-plans provide a means of overcoming the representational limits of existing plan formalisms. Defining domain knowledge about special cases and failure recovery via meta-operators provides an expressively complete approach, which obviates the need for special-purpose language extensions. This approach also expands the role that meta-planning plays in a planning architecture. The resulting transformations represent a special class of operations on plan networks: operations that reformulate a network rather than solve it. As an additional benefit, this approach eases the writer's task of providing thorough coverage of domain actions.

We are currently implementing the transformations as part of the GRAPE plan recognizer. This implementation uses Knowledge Craft™ and capitalizes on its facilities for context management, object schema, and integrated Prolog features. We are continuing to explore the role of deeper domain knowledge in planning systems, with particular emphasis on a deep model of programming process knowledge and its potential contributions in an intelligent assistant.

6.0 References

[1] Broverman, C.A., and W.B. Croft, "A Knowledge-based Approach to Data Management for Intelligent User Interfaces", *Proceedings of Conference for Very Large Databases*, 1985.

[2] Broverman, C.A., K.E. Huff, and V.R. Lesser, "The Role of Plan Recognition in Intelligent Interface Design, *Proceedings of Conference on Systems, Man and Cybernetics*, IEEE, 1986, pp. 863-868.

[3] Carver, N., V.R. Lesser and D. McCue, "Focusing in Plan Recognition", *Proceedings of AAAI*, 1984, pp. 42-48.

[4] Chen, P.P., "The Entity-relationship Model: Toward A Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, March 1976, pp. 9-36.

[5] Croft, W.B., L. Lefkowitz, V.R. Lesser and K.E. Huff, "POISE: An Intelligent Interface for Profession-based Systems", Conference on Artificial Intelligence, Oakland, Michigan, 1983.

[6] Croft, W.B., and L.S. Lefkowitz, "Task Support in an Office System", *ACM Transactions on Office Information Systems*, vol. 2, 1984, pp. 197-212.

Knowledge Craft™ is a trademark of Carnegie Group Incorporated.

[7] Huff, K.E. and V.R. Lesser, "The GRAPPLE Plan Formalism", Technical Report 87-08, Department of Computer and Information Sciences, University of Massachusetts, Amherst, 1987.

[8] Huff, K.E. and V.R. Lesser, "Intelligent Assistance for Programmers Based Upon a Formal Representation of the Process of Programming", Technical Report 87-09, Department of Computer and Information Sciences, University of Massachusetts, Amherst, forthcoming.

[9] Kaiser, G.E., and P.H. Feiler, "An Architecture for Intelligent Assistance in Software Development", *Proceedings of the Ninth International Conference on Software Engineering*, IEEE, 1987, pp. 180-188.

[10] Lehman, M.M., "Process Models, Process Programs, Programming Support", *Proceedings of the Ninth International Conference on Software Engineering*, IEEE, 1987, pp.14-16.

[11] Litman, D., "Plan Recognition and Discourse Analysis: An Integrated Approach for Understanding Dialogues", PhD. Dissertation (also TR 170), Department of Computer Science, University of Rochester, 1985.

[12] McDermott, D., "Planning and Acting", *Cognitive Science*, vol. 2, 1978, pp. 71-109.

[13] Nilsson, N.J., *Principles of Artificial Intelligence*, Tioga Publishing, Palo Alto, California, 1980.

[14] Osterweil, L., "Software Processes are Software Too", *Proceedings of the Ninth International Conference on Software Engineering*, IEEE, 1987, pp.2-13.

[15] Sacerdoti, E.D., *A Structure for Plans and Behavior*, Elsevier-North Holland, New York, 1977.

[16] Stefik, M., "Planning and Meta-planning", *Artificial Intelligence*, vol 16, 1981, pp. 141-169.

[17] Tate, A., "Project Planning Using a Hierarchical Non-linear Planner", Dept. of Artificial Intelligence Report 25, Edinburgh University, 1976.

[18] Wilensky, R., "Meta-Planning: Representation and Using Knowledge About Planning in Problem Solving and Natural Language Understanding", *Cognitve Science*, vol. 5, 1981, pp. 197-233.

[19] Wilkins, D.E., "Domain-Independent Planning: Representation and Plan Generation", *Artificial Intelligence*, Vol. 22, 1984, pp. 269-301.

[20] Wilkins, D.E., "Recovering From Execution Errors in SIPE", TN 346, SRI International, January, 1985.

This work was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under contract No. F30602-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

Technical Issues for Performance Estimation *

Allen Goldberg
Kestrel Institute†

July 9, 1987

1 Objectives of the Performance Estimation Assistant

This paper will report on the plans and progress toward the construction of prototype Performance Estimation Assistant (PEA). The objectives of the PEA is clearly stated in the KBSA report [2]. We include the relevant section from that report:

> The long-term goal of the performance facet is to help to create and maintain efficient programs that meet their performance requirements. The performance facet will guide performance decisions at many levels from requirements specifications to very-high level programs to low-level code. Performance assistance capabilities are critical for making practical tools of very-high-level, executable specification languages. Because the key disadvantage of such specifications is their lack of efficiency when executed straightforwardly, the important factor in their utility is being able to find efficient implementations. During development, efficiency estimation will be used to predict and compare the costs of proposed alternative data structure choices. With this capability, either a programmer or an automated program synthesizer can select a data structure. KBSA will also give performance advice about what control structures to use, what optimizing transformations to apply, and what algorithms to use. Thus, program analysis includes not only data flow and control flow analysis, but also higher-level analysis, such as algorithm analysis, to determine the time and space efficiency of programs, to suggest modularizations, and to find bottlenecks. It also involves augmenting application domain models to include some cost information. At the requirements level, advice will be given about the relative costs of different proposed features.

*This research was supported by the Rome Air Development Center (RADC) under contract F30602-86-C-0026. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied of RADC or the U.S. Government.

†1801 Page Mill Road, Palo Alto, CA 94304.

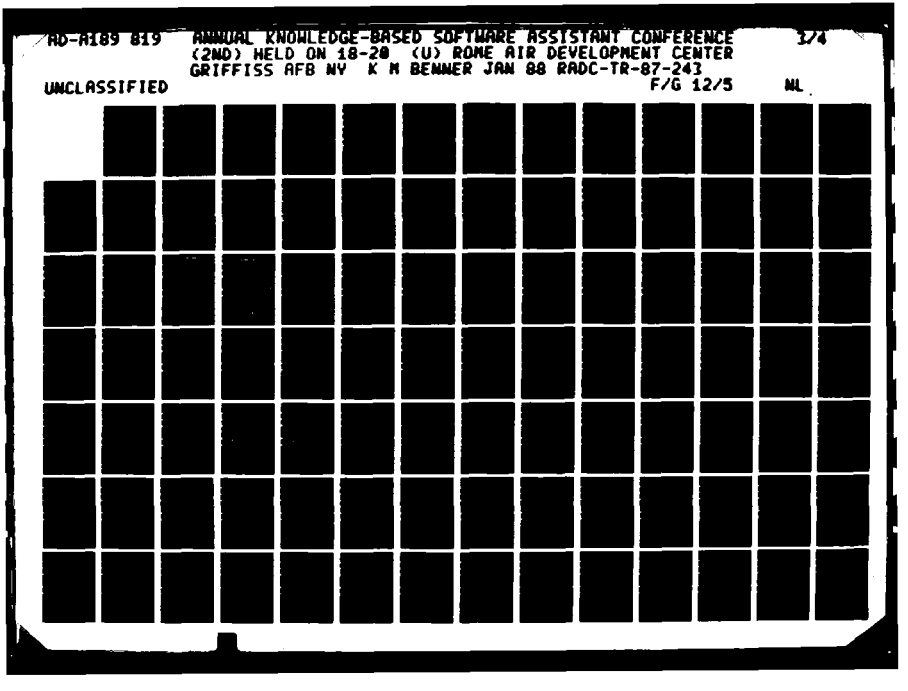
We can see from this description that the primary function of a PEA is to select among alternative implementations of a specification. Thus it must be closely coupled with the development facet, which generates these alternative implementations. Although one can imagine building a PEA which is in some sense generic, in that it does not depend at all on the generator, i.e a specific language and associated transformations, this will not be the approach we take. Our work has indicated that a useful PEA must embody knowledge of the space being searched, as do heuristic functions which control the search in classical AI problems. Thus the prototype PEA is specialized to a particular development task, data structure selection. This consistent with the short-term goals described in the report. Two of three short-term goals are (again quoting from [2]):

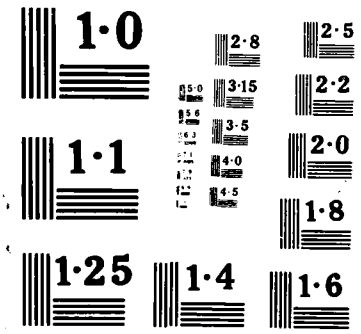
> **Symbolic Evaluation** Symbolic evaluation is a basic analysis technique that is useful in many of the KBSA facets described. However, it is crucial for the performance facet. The performance facet needs to be able to propagate and integrate efficiency estimations and to perform symbolic analysis on partial specifications.
>
> **Data Structure Analysis and Advice** A short-term target for the performance facet will be a set of estimators for data structure selection that are reasonably robust when handling conventional data structures (probably excluding external memory devices). These estimations could be used for automatic data structure selection or for advice to manual implementors. Efficiency estimation activities will be limited to those necessary for data structure selection, including the use of both rules of thumb and heuristic algorithm analysis. As an initial target, efficiency estimation will provide approximate, average-case performance analysis. The agents will compute and transform annotations about efficiency characteristics as programs are transformed, and will record cost analysis decisions for the benefit of future users. Some bottleneck finding also should be feasible in the short term; it is valuable for both automated and manual systems, and is a fairly straightforward extension of the basic performance analysis capability. By limiting the performance facet initially to data structure selection advice, we take a conservative position and increase the likelihood of success. It may be necessary, if certain applications are undertaken, to include other optimization decisions.

The first of these items is a method of analysis, the second specific functionality. We are using a symbolic evaluation capability to assist data structure selection, and designing the symbolic evaluation package in a *reusable* way. Thus when more sophisticated PEA's are constructed, this package will not have to be rewritten.

Specifically, *data structure selection* is the selection of of *efficient* ADA-level representations for set-theoretic data types. Finding efficient representations for set-theoretic data types is an [illegible] performance estimation problem because:

- There is high likelyhood of integration of this system with other KBSA [illegible]
- It is intrinsically an important problem to solve. This is [illegible] about programming. For example, university courses [illegible] part of a computer science curriculum


AD-A189 819
ANNUAL KNOWLEDGE-BASED SOFTWARE ASSISTANT CONFERENCE (2ND) HELD ON 18-20 (U) ROME AIR DEVELOPMENT CENTER GRIFFISS AFB NY K M BENNER JAN 88 RADC-TR-87-243
1/4
UNCLASSIFIED
F/G 12/5
NL


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0

- A generator for the space of data structure implementations exists. This is one of the few problems for which a good generator of alternative implementations exist.

As part of the project an initial prototype data structure selector has been constructed. It has provided evidence that good results will emerge from this project.

2 Approaches to Performance Estimation

The development of code from specifications can be simply modeled by a tree structure where nodes represent partially implemented specifications and arcs represent transformations. Paths through the tree represent derivations where the final nodes represent executable target-language programs. A crucial role of the PEA is help guide the search through this tree. The alternative programs that can be derived are distinguished mainly on the the basis of their resource consumption—they all implement the same functionality (since transformations are correctness-preserving).

The PEA must perform analysis of the program to guide the search. This analysis, in its details depend on the specific form of specifications and programs as well as the specific generator for the space. For example, because functional programming languages have clean, theoretical properties. such as referential transparency, and no notion of control flow, performance analysis is simplified in this context. In the next section we will provide specific details of the generator and the specific analysis requirements. Generally, we are able to classify approaches to this analysis that is not specific to the generator employed. Briefly they are: the *qualitative approach*, *symbolic approach*. *simulation-based approach* and the *statistical approach*. The reader is referred to the accompanying paper in this volume for more detail. The PEA we are building will utilize the qualitative and symbolic approaches.

3 The Generator

In this section we first describe the *generator* that the PEA will use by describing the specification and programming language and the transformation rules which generate the space of alternative implementations for the language. The goal of this work is to produce a PEA, but as explained above, the tight interaction between development and performance requires that a specific generator be used. Our approach is to incorporate an existing generator into the delivered system. Thus we will minimize effort not directed specifically at performance estimation.

The specification language, which is called PERFORMO, is a functional language with high-level set-theoretic data types. Our intention is not to produce a polished, practical language; that is a distraction from the goals of our work. Rather the language provides a minimal framework on which to construct a data structure selector. The transformation rules apply to the data structures of the program only, and will be directed by the PEA to select efficient, ADA-level data type implementations. The object program will remain at the functional level. That is, we are not concerned with control structure optimizations, such as recursion-to-iteration constructs.

The advantage treating the data structure selection problem in a functional context is that the analysis on which data structure choices are made is greatly facilitated because of the referential transparency of declarative-level forms. Generally assertions about program properties that are true universally, rather than true at certain program points can be inferred without the need for elaborate control flow analysis. The analysis techniques developed could be extended to work on imperative programs if desired. The data structure selection is based on an analysis local to each function definition. This restriction can be removed in a larger-scale prototype.

3.1 PERFORMO

PERFORMO is a *single-assignment functional language*. Other languages of this variety are VAL [3], developed at MIT, and SISAL [4], developed at Lawrence-Livermore Lab. In such a language each "variable" is defined by an equation which is computed only once (hence the notion of single assignment). These language preserve the referential transparency that enhances the analyzability of functional languages, while providing a convenient syntax.

Unlike other functional languages which primarily rely on lists as the fundamental data type, PERFORMO provides very-high-level, set-theoretic Data types and lower-level data types which the set-theoretic types are refined into. Another notable feature of PERFORMO is its flexible iteration constructs, which are useful for refining many of the operations defined on set-theoretic objects. Many operations which are implemented by explicit recursive calls in functional languages without this construct can be more simply implemented by the iteration construct.

3.1.1 The Data types

PERFORMO is strongly-typed and provides set-theoretic type constructors. As part of a declaration the user may specify assertions about the defined data objects. These assertions can be used to define sub-types, provide information as to the maximum size of the data objects, and other properties. These assertions provide information that is used by the PEA to select implementations. At the same time these assertions provide a natural, high-level specification capability to the user. For example, the type map integer to integer may be restricted to map S to integer where S is a subset of the integers defined assertionally in the program. As we shall see, this assertion is supplying *containment information* which is basic analysis data for data structure selection.

The Base Types The base data types of the language are *integer*, *real*, *Boolean*, *char*, and *atom*. The usual operations are defined on these types. In particular, an if-then-else construct is defined.

The Composite Types The composite types are *sets*, *maps*, *sequences*, *products*, and *relations*. Thus the types provided are similar to those in REFINE, SETL [6] and AP5[1]. An important difference however, is that the notion of a *normal form* is introduced. Programs may be written using a large collection of useful operations on these types, but the programs will be

mechanically transformed into programs using just a few operations. Such programs are said to be in *normal form*. Our analysis capabilities are based on programs in this restricted form, providing an important simplification.

We briefly review the operations in the normal form language. The operations found in the above mentioned languages can be easily expressed in terms of these primitives. Many of the powerful operations on composite objects, such as reduction operations, quantification, and set-formers require iteration over a composite structure to implement. The language contains one general form of iteration called a *generator expression*. In the normal form all these familiar operations are reduced to generators. The syntax of generators will not be given here, but reserved for the complete language description.

Sets Sets are unordered collections of non-repeating elements. The operations on sets are: explicit set former (create a set with the specified elements), less (create a set from a given set with one element removed), with (create a set from a given set with an additional element inserted), arb (choose an element randomly from a set), $\in$ (a predicate which tests set membership), and union (form the union of two sets).

Maps Maps associate with elements in a domain set, elements from a range set. The operations on sets are: lambda definition (define a map with a lambda definition), domain (return the set composed of the domain of the map), range (return the set composed of the range of the map), and application (compute the range element corresponding to a given domain element).

Sequences Sequences are ordered collections of elements. The operations are: explicit sequence former (which creates a sequence out of the given elements), $\in$ (a predicate which tests membership in a sequence), indexed retrieval (returns the element at a specified position in a sequence), and insertion.

Products Products are like Pascal records. The operations are product formation and component selection.

Relations Relations are simply sets of products. Thus all the set operations may be applied to relations as well as projection along any component.

3.1.2 Function Definitions

A functional program in a single-assignment language consists of function definitions composed of a header which defines the input parameters and the output value of the function and a body consisting of declarations, assertions and defining equations for each variable in the function. We call these equations *definitions*.

3.2 Refinement Rules

The generator of the space of data structure implementations are called *refinement rules*. A refinement is a collection of rules that refine the variable declaration of an object along with all operations performed on the object. Refinements are designed so that they may be composed together in order to create the implementation of the variable. This allows a stepwise refinement approach to data structure selection.

Here, briefly is a description of some of the refinements used in the PEA.

set-to-sequence A set of objects is represented as a sequence of those objects. Each object appears in the sequence once and the order is arbitrary. Implementations based on maintaining the sequence in sorted order are not considered.

set-to-characteristic-function A set is represented by a total map from some superset of its domain to Boolean. The superset must be specified in the rule application and must be a variable that appears in the program or an integer (or character) subrange.

map-to-array A map from an integer or character subrange is represented as an array.

map-to-field-of-product Applies only to an non-parameterized object, that is, one for which there is only one instance of the map. An example of a parameterized object is the type of elements of the range of a map. There is one such element for each domain element of the map. Furthermore the product must represent a set which is a superset of the domain.

map-to-relation A map is represented by its graph.

seq-to-list A sequence is represented as a list.

seq-to-map A sequence is represented as a map from an integer subrange to the element type of the sequence.

relation-to-map A relation is represented as a map from one of its fields to a set of tuples of the remaining fields.

relation-to-set-of-product The relation is stored as a set of products.

object-to-accessor An object here is a variable of any type, including integers, reals, Booleans. We require a base set S such that the object is an element of the base. Furthermore S must be represented so that a pointer or array reference accesses its elements.

accessor-to-pointer Access is via a pointer.

accessor-to-integer-subrange Access is via an array index.

We illustrate how these refinement rules can be composed together to implement a set-theoretic data type at the ADA level. Suppose that a variable of type map appears in a program. Using the refinement map-to-relation the variable may be refined into a relation. Using relation-to-set-of-product, the relation may be represented as a set of products. The set may may then be

represented as a list using the refinement set-to-list. Note each time a refinement is used operations are refined as well. For example, if the operation 'domain', was applied to the original variable then this operation will be refined into code which enumerates the list, extracts the first element of each product and collects them into a set.

Notice that other refinement paths are possible, leading to a tree-shaped space of possible implementations. The PEA must select the refinement path that leads to an efficient representation.

4 The Performance Estimator

The PEA can be broken into two parts. The first part is called the *analyzer*, the second the *selector*. The analyzer is responsible for collecting information which will provide the basis for the decision making by the selector. The source of this data is the program, and annotations (assertions) in the program written by the user.

One advantage of working in the domain of data structure selection is the existence of criteria for determining when an efficient representation has been achieved. Namely, we strive for simple representations in which every data structure operation is implemented by its theoretic lower-bound. For example we insist that map application be performed in constant time and that enumerations of a set take time proportional to its size. Since our generator can only generate relatively simple structures, the constant factors associated with these operations will be low. For example hash table representations are not included because hashing while constant time (on average) is too complex to be efficient. If all operations are implemented within its theoretic lower bound then the data structure selector has succeeded in finding the best possible implementation. Furthermore, the system can report that the best implementation has been found. Often this is possible.

4.1 Analysis

The data collected by the analyzer is both of a qualitative and quantitative nature, used for *qualitative performance estimation* and *symbolic performance estimation*, respectively. The qualitative analysis will be sophisticated, similar to the analysis that an optimizing compiler performs.

The analysis data is derived by *inspection* (of the program and user-supplied assertions), *symbolic evaluation*, and *inference*. As an example of the use of inference suppose the size of a a set S is known to be n, f to be a one-one function and T is defined by the equation $T = \{f(s) : s \in S\}$, then the size of T can be inferred to be n. Inference of this variety is easily expressed as transformation rules in REFINE, the system used to construct the PEA. The inference needs of the system are being restricted so that a forward inference strategy will be efficient.

Below we summarize the kinds of analysis that will be performed, and indicate the technique used for the analysis.

4.1.1 Type Analysis

The language is strongly typed. A Milner-style type checker/analyzer as utilized in ML, REFINE and other languages will be employed. The construction of such a checker is well understood. This is done by inspection and inference (if the system infers types from context).

4.1.2 Containment Analysis

This refers to deducing subset and membership relations among the data objects of the program. We will also determine whether sets are disjoint as part of this analysis. This is done by symbolic evaluation.

4.1.3 Operation Analysis

Examine each *use* of a variable and determine what operations are performed on the variable. Data structures are selected to support efficient implementation of those operations which actually appear in the program. This is done by inspection.

4.1.4 One-one Analysis

One-one analysis seeks to determine if elements of sequences are unique and whether functions and relations are one-one. A fundamental issue in data structure selection is to support constant-time access to an element of a composite data structure. This is often difficult, and so a strategy of optimizing away the need for such access is important. For example when a value is inserted into a set a check whether the value is already in the set is required. If this check can be proven to be unnecessary more efficient can can be generated. One application of one-one analysis is to remove such checks and hence ease the task of the data structure selector. This is done by inspection and inference.

Related, but not nearly as useful is *few-to one analysis*. A map is *few-to-one* if only a constant number of domain elements map to the same range element. This information is useful for symbolic analysis of variable sizes.

4.1.5 Symbolic Analysis

In this analysis we first develop symbolic expressions that estimate the size of composite-valued variables. This information is then used to compute operator frequencies and ultimately complexity estimates on the time and space requirements of an implementation.

The analysis starts by assigning a symbolic variable to represent the size of each composite input value. Integer-valued program variables may also appear in these symbolic expressions. The user has the option of specifying a maximum bound on the size of an input variable, and then this

constant will be used instead. Then using the definitions of non-input variables given by the program this size information is propagated. The above described qualitative analysis aids this process, especially one-one and containment analysis. There are difficult technical issues that must be faced in deriving these estimates, which were considered in [5]

Given variable sizes we can estimate operation frequencies. Then if it is impossible to find a data structure representation that efficiently implements two operations both being performed on the same variable, these frequencies can be used to decide which operation should get the efficient implementation.

4.2 Selection

The fundamental difficulty of choosing data structures is to insure that element tests for sets and application for maps is performed in constant time. This is not achieved if a set is represented as a list. To test if an element is in a list the whole list must be searched, which takes time proportional to the size of the list. Similarly if a map is represented as a list of domain-range pairs, a similar search must be performed when applying a map to an domain element. However, suppose a set is known to be a subset of an integer subrange, for example $A \subseteq \{1, 2, \ldots, 100\}$. The A may be represented as a bit-vector, i.e. an array indexed by 1..100 of boolean values with the property that the ith element of the array is true if and only if $i \in A$. This will provide constant time element tests for A, but is restricted to subsets of integer subranges. This idea can be extended.. Suppose our analysis reports that $A \subseteq B$ and that $x \in B$. We need to test if $x \in A$. Then we may represent A as an additional Boolean-valued field in a product representing elements of B with the property that the Boolean is true if the element happens to be in A as well. x is represented as a pointer to elements of B. To test if $x \in B$ just follow the pointer x to the product representing the element and check the bit indicating whether the element is in A. This can be done in constant time. This is the basis of the data structure representations in the SETL compiler. Some of the analysis done by the PEA is to detect subset and element information (containment analysis) that is needed for selecting these implementations. This illustrates just one of the complexities of data structure selection. Others are dealt with by the PEA, for example minimizing the need to convert data representations in different portions of the program.

5 Implementation Status

The implementation of the PEA is progressing using version 2.0 of the REFINE system. Currently, the designs of most components are complete. The language can be parsed and its abstract syntax is defined. Analysis techniques have been design and are currently being implemented. Data structure refinement rules have been written. Remaining is the design and implementation of the selector itself, which will generalize the data structure selector used in our first prototype.

6 Future Work

In the present PEA a functional program written using high-level data types is transformed into a functional program operating on the low-level data type implementations. In our future work we will transform the control structures as well into low-level procedural code. The framework we have described will accommodate this extension. The enhanced prototype would incorporate efficiency knowledge in the following areas:

Finite Differencing Finite differencing is an important optimization that can lead to asymptotic improvements in program efficiency.

Proceduralization This is the application of subroutine and module decomposition device.

Algorithm Design Advice In particular the capability for constructing symbolic efficiency estimates of recursive programs will be established. This will require construction of a recurrence relation solver.

Reformulation Efficiency estimates can be used to guide transforms that redefine object definitions into an equivalent, but perhaps more efficient form. An important class of these reformulations implement store-vs-compute decisions.

Loop Fusion Operations over set-theoretic, composite objects are implemented with extensive use of iteration. These loops can often be combined to yield a more efficient program. We have extensively studied this optimization. The use of PERFORMO's iteration construct support the expression of loop combining transformations.

References

[1] Donald Cohen. Automatic compilation of logical specifications into efficient programs. In *Proceedings of the 1986 National Conference on Aritificial Intelligence*, AAAI, Philadelphia, PA. August 15-17, 1986.

[2] Cordell Green, David Luckham, Robert Balzer, Thomas Cheatham, and Charles Rich. *Report on a Knowledge-Based Software Assistant.* Technical Report KES.U.83.2, Kestrel Institute, July 1983.

[3] James R. McGraw. The val language: description and analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, January 1982.

[4] McGraw, James, et. al. *SISAL: Streams and Iteration in a Single Assignment Language.* Technical Report M—146, Lawrence Livermore Laboratory, March 1985.

[5] Xiaolei Qian and Douglas R. Smith. Integrity constraint reformulation for efficient validation. In *Proceedings of the thirteenth International Conference on Very Large Data Bases*, London, August 1987.

[6] Edward Schonberg, Jacob Schwartz, and M. Sharir. An automatic technique for the selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143, April 1981.

Designing an Effective User Interface for a Knowledge-Based Development Environment

Tim King
Honeywell Systems and Research Center
3660 Technology Drive
Minneapolis, MN 55418

Abstract

One of the chief factors in the success of transferring Knowledge-Based Software Assistant (KBSA) technology to industry will be the ability of users to interact efficiently and effectively with the KBSA system. As a result, KBSA's User interface must be carefully designed to provide a wide variety of users with easy access to a wide variety of information. This paper describes design issues of an effective interface, its required functionality, and possible approaches to providing that functionality.

1 Introduction

An effective User Interface (U/I) plays a crucial role in any system. This is especially true of the Knowledge-Based Software Assistant (KBSA). Because of the wide range and amount of available information and the varying degrees of user sophistication, a powerful and flexible U/I will be critical in the acceptance and success the KBSA. The KBSA knowledge base will contain project management information such as schedules, available resources, resource allocation, costing data, project structure, and policy. It will contain technical information such as requirements, specifications, code, test data, and test results, as well as information about the KBSA itself. KBSA users will span the spectrum from the technically expert to technically naive. The KBSA must accomodate each kind of user's request for whatever type of assistance, access, or information they require. There appear to be at least five areas in which the U/I will play a role:

1. Runtime support,
2. PMA,
3. Help/Tutor,
4. Query,
5. Inspection.

First, we will consider the design guidelines for constructing an effective U/I. Then, we will consider possible approaches for providing the functionality of the above five U/I areas with respect to the attributes of an effective U/I.

2 Designing an Effective User Interface

An effective U/I, like any well designed system, should be based on sound principles or standards. Following such standards provide at least three benefits:

1. Better interface designs,
2. Increased consistency across various systems,
3. Reduced cost in system implementation.

Although much progress has been made towards defining these standards [9], a definitive set of U/I standards is still a long way off [8]. However, even though we currently lack standards, many attributes have been identified that are fundamental to an effective interface [11,17,19].

2.1 Desirable User Interface Attributes

First, the interface should provide **consistent** and **intuitive** access to the system. As users interact with the KBSA, they will do so primarily by applying methods to objects. They will edit objects, compile objects, select tasks, etc. A consistent interface will enforce certain restrictions. If a method is applicable to various objects, then the manner in which it is applied will be the same for all objects in all situations. For example, suppose users can edit both requirements and specification objects. It is inconsistent for the requirements editor to be invoked via a command line interface while the specification editor is invoked via mouse selection. Further, interaction with the editor should be consistent in both cases. Deleting a line or moving text should appear the same whether editing requirements or specifications. Help and query facilities providing accurate, up-to-date information should also be consistently available. For example, a help facility could be available to users at any time by pressing a "HELP" key. After requesting help, users can ask assistance on any topic(s), then return to whatever task they were performing prior to requesting help. Such consistency will play a big role in the interface's intuitiveness. An intuitive U/I allows users to accomplish tasks in "natural" ways, by asking themselves "how would I like to tell the computer what I want?". Given an intuitive U/I users usually would be able to correctly guess the thing to do.

For example, it is more intuitive to say "EDIT filename" than "INVOKE SYS.EDITOR filename".

Second, the interface should be **friendly** [11]. That is, it should provide unobtrusive access to the system which puts users at ease. Such an interface must be cooperative, preventive, and conducive. A cooperative interface actively assists users by keeping them aware of basic capabilities, allowing them to explore additional capabilities easily, and keeping them informed on what the system is doing on their behalf and what progress is being made. A cooperative interface will also allow users to suspend tasks and quickly return to them at some later time. A preventive interface must be designed to expect mistakes, make allowances for mistakes, and allow users to easily recover from mistakes. Finally, a conducive interface should be reliable, predictable, and it should encourage users to explore new capabilities without fear of irretrievably destroying vital information.

Third, the interface should provide access to intelligent **help** and **query** facilities [17]. Studies have shown that such on-line assistance can substantially affect a system's effectiveness, usability, and learnability. Users require easy access to help information describing features and functionality of KBSA components such as the Requirement Assistant, the Specification Assistant, editors, and compilers. Users also require easy access to data in the knowledge base such as available resources, schedules, and task assignments.

Finally, since users have diverse backgrounds and varying degrees of technical knowledge, the interface should have some mechanism for **user modeling** [19]. This includes determining and recording a user's beliefs, goals, plans, and knowledge in an attempt to tailor the system's interaction with the user. In doing so, the system can more appropriately determine what kinds of information to supply the user, it can more actively assist the user in achieving their goals, and it can detect inappropriate plans for achieving those goals.

2.2 User Interface Design Principles

Many approaches can be taken to satisfy these requirements for an effective user interface. In determining which approach is appropriate in each situation one should attempt to follow design principles such as those described in [4,15]. In [4], Gould and Lewis suggest four guidelines:

1. Focus on users early to determine their needs and capabilitites.
2. Use an interactive design process in which potential users are part of the design team.
3. Attempt to empirically measure the usability and learnability of the interface.

4. Use an iterative design process in which suggested changes or enhancements are incorporated in the next design.

In [15], Rich suggests seven factors to consider in determining how to provide required functionality:

1. Cost of the interface. Different kinds of interfaces (menu, command-line, graphical, natural language, etc.) require differing amounts of effort in implementation and maintainence. The cost must be balanced against the required functionality.

2. Ease of learning. Some interfaces (menu, natural language) are easier to learn than others. The interface complexity must be geared toward the user community.

3. Conciseness. Some interfaces (menu, keyword) are more concise and require less interaction to communicate something. The conciseness/verbosity of the interface must correlate with the tasks to be performed.

4. Need for precision. Some interfaces (natural language) are less precise than others. The precision of the interface must match the precision required by the task.

5. Need for pictures. In many cases, pictures communicate information much better than text. The interface should be capable of displaying pictures whenever appropriate.

6. Semantic complexity. The required complexity of the interface is directly proportional to the complexity of the task being performed. The interface must balance its own complexity against that of the tasks it will be used to perform.

7. Promising more than can be delivered. In most cases, if a system has a complex interface, users will expect complex behaviour from the underlying system. The interface must not mislead users by shrouding a simple system in a complex interface. Interface designers must be careful to balance the interface's complexity against the complexity of the underlying system.

By following these guidelines and keeping in mind the desirable U/I attributes, an effective, helpful, and easy to use U/I can be constructed for the diverse KBSA user community which meets all of the above requirements and provides the appropriate mix of functionality and complexity.

3 User Interface Architecture

This section describes the U/I functional requirements of the KBSA system and discusses possible approaches in each area. Currently we have identified five areas in which U/I plays a role. These areas are: a runtime support environment, a PMA interface, a Help/Tutor facet, a Query facet, and an Inspector facet. Each of these areas is considered in turn, along with possible approaches to providing the capabilities described.

3.1 Runtime Support Environment

The runtime support environment provides a set of routines available for use in constructing user interfaces for the various facets. It should include routines for window management, icon definition, user input, menu definition, mouse sensitivity, program output, etc. Input routines should support both command-line (textual) and mouse-driven input. Output routines should support both textual and graphical display of information. Since standards are beginning to emerge in this area, one of the proposed window management standards, such as [7], would be adequate to support these capabilities.

The runtime support environment should also specify a set of candidate interface standards/conventions which facets should follow in their interactions with users. As mentioned earlier, no such standards exist. However, the KBSA should specify some conventions for user to facet interaction based on the desirable U/I attributes described earlier. This will provide for increased consistency across facet boundaries, and will not force users to learn different interface conventions for different facets. Of course, these standards should evolve as more is understood about U/I in general and KBSA U/I in particular.

3.2 PMA Interface

The PMA Interface provides the link between users and the PMA. This interaction will occur primarily whenever project policy cannot autonomously determine which action to perform next. For example, suppose a developer finishes the first of three assigned tasks. Further, at that point, policy dictates that they begin working on any remaining tasks they are responsible for. However, since two tasks remain, the system cannot determine which task the developer should begin next. The developer must decide and inform the system which task to select. As another example, suppose a developer successfully finishes testing a specification. At that point, policy allows the developer to go back and modify the requirements associated with the specification, or to submit the specification and the test results

for review. Again, the system cannot determine what action to take next based on this policy. The developer must decide.

In such situations, the system would invoke the PMA Interface. First, this interface would prioritize the alternative actions based on PMA information such as scheduling, resource availability, and known priorities. Then, it would present the user with the prioritized alternatives. To provide this functionality, a simple menu would be adequate. Higher priority options could appear at the top of the menu while lower priority options would appear at the bottom or, each alternative could be explicitly prioritized.

This will provide the basic functionality required of the PMA Interface. However, in the long run, some sort of explanation mechanism will probably be required. Users will want to know why the PMA recommends that they proceed with some task while postponing others; or, they may want to validate its reasoning in weighting their tasks if those tasks are critical to a project. Several such systems exist [18.2.5]. It is uncertain whether this functionality should reside in the PMA or in the PMA Interface, but in either case, the appropriate subsystem could incorporate an explanation mechanism based on some form of simple command language (or a short list of mouse selectable commands) for requesting various types of explanation (e.g. EXPLAIN, VERIFY, etc.). Responses could be based on simple, built-in answers to this limited set of questions. However, this form of interaction would probably prove too restrictive. In order to provide a truly effective, broad-range of interaction, a more flexible and powerful form of communication would be necessary. Thus, the explanation mechanism could be based on Natural Language (NL). In such a system, users could request explanations and verifications of the PMA's reasoning in NL and they would receive responses in NL generated for the specific situation.

3.3 Help/Tutor Facet

The Help/Tutor facet provides an intelligent help facility to the users of KBSA. It will provide explanations of how to use individual facets and the KBSA as a whole. To provide this functionality, as new facets or tools are integrated into the KBSA, knowledge about their features and use also must be integrated. The Help/Tutor facet will use this information to describe components to the user. For, example, suppose a developer is working on a KBSA system on which a new editor has been installed along with knowledge of its features and functionality. Now, if the developer wishs to learn how to use the new editor, they merely invoke the Help/Tutor facet and request an explanation of the editor. At that point, they may ask for information such as how to save a file, how to delete a line, or how to move a block of text.

There are at least four options for providing this functionality: keyword, menu-driven, document browser, and NL. First, in a simple keyword approach, users enter a special command, such as "HELP", followed by keywords, such as a command name, options, or parameters [10]. The main advantage of this approach is its simplicity of implementation. The main disadvantages are that it can be too restrictive, it forces users to know which terms are allowed in constructing a request, and many replies are typically cryptic and uninformative. A similar, though less restrictive technique is described in [16], which uses an ELIZA-based parser to extract keywords from NL inputs. These keywords are then used to match substitution patterns from which an NL response is constructed.

Second, in a menu-driven approach, the system displays menus offering additional information [3]. As with the keyword approach, the main advantage of this approach is its simplicity of implementation. Its main disadvantages are that it is even more restrictive and inflexible than the keyword approach, and again, many replies are typically cryptic and uninformative.

Third, in a document browser approach, users can browse through online documents by referring to chapters, sections, subsections, or paragraphs [13]. Again, the main advantage of this approach is its simplicity of implementation. The main disadvantages are that the replies are only as good as the underlying documentation, and again, software documentation is notorious for being cryptic and uninformative. Further, depending on the underlying structure of the document, the amount of time required to find the desired information may be large. However, if the documentation is broken up into many small pieces and stored as a highly interrelated network [12], users can quickly cross-reference information, the system can avoid duplication overheads, and related information can be inferred from well-defined network relationships.

Finally, in a NL approach, users can request and receive help in NL [20]. The main advantages of this approach are that users need not learn a command language. NL is very flexible and powerful, responses can be tailored to individual users, and the implementors have greater leeway in providing additional features to help users clarify confusing responses. The main disadvantages are that effective NL interfaces are generally more expensive and much more complex than other interfaces, and NL queries tend to be more verbose than other types of queries.

3.4 Query Facet

The Query facet provides users with a facility for accessing the knowledge base. It will allow them to formulate queries (requests for information), translate these queries into a query format recognizable to the underlying

knowledge ba
base, and
could find out w
are assigned
They
allocation in
the status of
are developing

As a basis for
compose
the reques
are at least
approach and a

In tradi
a program
what info
the data stored in
form of interac
data in the database
language.
commonly
and column format.
of interface is
disadvantages are
few users are likely to
are likely to ask, th

In a NL approach
questions and
the NL queries and
tional syst
NL response to
is that users do
questions are muc
approach is that N

3.5 Inspector Facet

The Inspector facet
and modifying the
allow users to
In doing this, a user
this object,
the object's state

knowledge
manager
resources
ask for
they
users to
that satisfies
There
non-NL
essentially
determine
terms of
language. This
kinds of
query
databases
in a row
at this form
The main
use which
HCSA users
very complex.
They ask
takes
use of tradi-
ta and forms a
This approach
and complex
advantage to this
improvement.

viewing
This facet will
set states.
displaying
etc),
user has

to other objects (essentially a special kind of slot and value) [illegible] this information, they then can traverse the relationships and inspect further objects, thereby walking through the knowledge base. In addition, users will be able to modify a displayed object's state. They will have access to methods which allow them to assign values to the object's slots. They will also have access to the methods which are defined on the object (i.e., they are part of the object's class definition). The Inspector facet will provide a valuable debugging tool for both KBSA users and for KBSA system developers.

4 Summary and Conclusions

Given the above five areas of U/I support, KBSA users will have an effective interface to the KBSA. They will have access to FMS information and on-line Help facilities, and they will have a knowledge base query capability. KBSA system developers will benefit from the runtime support environment and the Inspector facet. The runtime support environment will provide a common pool of screen/window management routines, and the accompanying standards/conventions which will provide a fairly stable basis for interface development.

However, even with this kind of support, users may still reject KBSA. U/I designers must be careful to concentrate on providing an interface which possesses the desirable attributes of consistency, intuitiveness, friendlyness, helpfulness, and user modeling. They must also be careful to follow good design principles, such as those described earlier. Finally, U/I designers must remain open to new ideas, so that as research provides further insight into what constitutes a good U/I, they will be willing to try those ideas and incorporate the best of them.

References

[1] R. Cullingford, M. Krueger, M. Selfridge, and M. Bienkowski, "Automated Explanations as a Component of a Computer-Aided Design System", IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-12, No. 2, March/April 1982, pp. 168 - 181.

[2] R. Davis,"Interactive Transfer of Expertise", Artificial Intelligence, Vol. 12, No. 2, August 1979, pp. 121 - 157.

[3] R. Demers,"System Design for Usability", CACM, Vol. 24, No. 8., August 1981, pp. 499 - 500.

[4] J. Gould and C. Lewis,"Designing for Usability: Key Principles and What Designers Think", Proc. CHI'83, December 1983, pp. 50 - 53.

[5] D. Hasling, W. Clancey, and G. Rennels,"Strategic Explanations for a Diagnostic Consultation System", Int. J. Man-Machine Studies, Vol. 20, No. 1, January 1984, pp. 3 - 20.

[6] G. Hendrix, E. Sacerdoti, D. Sagalowicz, and J. Slocum, "Developing a Natural Language Interface to Complex Data", ACM Trans. on Database Systems, Vol. 3, No. 2, June 1978, pp. 105 - 147.

[7] IntelliCorp,"IntelliCorp Common Windows Users Manual", October 1986.

[8] G. Lynch and J. Meads, "In Search of a User Interface Reference Model: Report on the SIGCHI Workshop on User Interface Reference Models", SIGCHI Bulletin, Vol. 18, No. 2, October 1986, pp. 25 - 33.

[9] K. Lantz, "On User Reference Models", SIGCHI Bulletin, Vol. 18, No. 2, October 1986, pp. 36 - 42.

[10] R. Marcus,"User Assistance in Bibliographic Retrieval Networks Through a Computer Intermediary", IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-12, No. 2, March/April 1982, pp. 116 - 133.

[11] J. Meads,"Friendly or Frivolous?", Datamation, April 1, 1985, pp. 96 - 100.

[12] T. Nelson, "Literary Machines", T. H. Nelson, Swarthmore PA, 1981.

[13] L. Price,"Thumb: An Interactive Tool for Accessing and Maintaining Text", IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-12, No. 2, March/April 1982, pp. 155 - 161.

[14] Relational Technology Inc.,"INGRES/QUEL Self-Instruction Guide", January 1986.

[15] E. Rich,"Natural-Language Interfaces", IEEE Computer, September 1984, pp. 39 - 47.

[16] S. Shapiro and S. Kwasny,"Interactive Consulting via Natural Language", CACM, Vol.18, No. 8, August 1975, pp. 459 - 462.

[17] N. Sondheimer and N. Relles,"Human Factors and User Assistance in Interactive Computing Systems: An Introduction", IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-12, No. 2, March/April 1982, pp. 102 - 107.

[18] W. Swartout,"XPLAIN: A System for Creating and Expiaining Expert Consulting Systems", Artificial Intelligence, Vol. 21, No. 3, September 1983, pp. 285 - 325.

[19] W. Wahlster and A. Kobsa,"Dialogue-Based User Models", Proc. IEEE, Vol. 74, No. 7, July 1986, pp. 948 - 960.

[20] R. Wilensky,"Talking to UNIX in English: An Overview of an On-line UNIX Consultant", AI Magazine, Vol. 5, No. 1, Spring 1984, pp. 29 - 39.

Knowledge Based Tools

for Knowledge-Based Systems

Lance A. Miller, Ph.D
IBM Federal Systems Division
Artificial Intelligence Center
Gaithersburg, MD 20879

To be presented at the Second Annual Knowledge-Based Software Assistant Conference (KBSA), Utica, New York, 18-20 August, 1987.

Conference sponsored by the Department of the Air Force, Headquarters, Rome Air Development Center, Griffis Air Force Base, New York.

OVERVIEW

The focus of the Knowledge-Based Software Assistant Conferences is on advanced automated software development tools, and this year's theme is Technology Transfer. While adhering to these concerns, this presentation departs from the more typical tool assumptions in that it is concerned with the development of knowledge-based tools and procedures primarily for knowledge-based software systems.

Our concern for such tools derives from the emergence of a new business area in which customers' significantly large problems appear not to be satisfiable by traditional software and practices. These problems, while involving traditional system integration and communication issues, have at their core a large expert-system component requiring AI methodology and technology, at least for the development -- if not the delivery -- system. A critical requirement common to most of these problems is that the expert-system piece can seldom stand alone as a kind of consultant environment outside of and in remote communication with the customer's system. Rather, the AI software must be embedded within the total software environment, to be compiled, called, used, and banished just like any other specialized software package or utility. This requirement is not only a technical one -- how to implement an embedable expert-system -- but it is also an administrative one in that the AI component must be manageable like other software susbsystems with reviews, tests, configuration management, etc.; and for government customers this means putting the AI piece under the aegis of formal -- and formidable -- military standards (e.g., *MILSTDs 490, 483, and 2167*). Reasonable as this may be, the major difficulty of complying is that there is so little experience with AI systems, particularly large ones, that there is barely the notion of a system life-cycle much less a consensus; without a well-specified life-cycle there is little basis for scheduling reviews, for costing, for documentation, or generally for any of the typical management activities which insure the successful completion of large software projects.

This presentation overviews three aspects of our approach to remedying this situation: (a) a proposed life-cycle for Knowledge-Based Systems (KBSs); (b) techniques for knowledge-base verification, and (c) representation and traceability of Requirements.

A PROPOSED KBS LIFE-CYCLE

If one wanted to emphasize the differences between KBS and traditional systems, one might choose to represent their respective life-cycles as shown in Figure 1. Not only are the stages dissimilar but so also are the operations that are performed to move from stage to another. Without discussing this contrast fully we note that each life-cycle does a different thing best: for the

traditional waterfall method it is the traceability of requirements, but for the KBS method it is the visability of the Operational Concept that is most striking.

We believe that Figure 1 is a useful and defensible view of the contrasts among many present-day KBS and traditional life-cycles, especially during development. However, for the longer view, which takes in transitioning from development system to delivery system, through integration, out into the far reaches of maintenance, we propose a different pair of life cycles, as shown in Figure 2. These emphasize the essential similarities in the building, fielding, and maintaining of all large complex systems -- whether they are KBS in an AI language or traditional systems with procedural code. Here the successive stages are the same in both cases; it is presumed that the requirements are not perfectly understood for both, hence the need for a development system to support prototyping prior to a probably-rewritten delivery system. Test and integration is placed after the delivery system phase to emphasize that the system being developed by these life-cycles is part of some larger complex system. The key distinction is that for KBSs, the Knowledge Acquisition phase -- comparable in some respects to Traditional's Requirement Specification phase -- is replicated to a limited extent for all phases of the KBS cycle, as shown by the vertical column connecting to each.

We call this view the Elevator Model to emphasize the top-level importance of Knowledge Acquisition for KBS products whereby difficulties in all succeeding phases can be addressed by initially spending considerable time in knowledge acquisition anticipating problems in subsequent phases and then responding to the remaining inevitable problems in successive phases by essentially bringing them up to the Knowledge Acquisition phase and reworking the knowledge. This view was developed primarily in the interests of assuring the verifiability of knowledge-bases, and maximizing their validity, particularly for systems not easily accessible after deployment, and it was presented at length in a recent NASA conference (Miller, 1987).

The means by which the Elevator Model achieves its goals is to elaborate the initial acquisition phase into three pairs of knowledge-development activities, as shown in Figure 3; each first element of the pair is tightly coupled to an intensive testing partner, and the three pairs are somewhat more loosely coupled to each other. The first of these stages is Knowledge Acquisition per se, and we are exploring a variety of techniques to facilitate and even automate this process. The second stage is to amass, coordinate, and transform relatively "raw" acquired knowledge into a common formal representation system suitable for verification testing; details of this stage will be covered in the next two sections of this present overview. The third stage is to map the formal knowledge into whatever rule syntax is used by the expert-system shell for prototyping; in the case of IBM's ESE (Expert System Environment) product, this means converting the declarative knowledge base into ESE Parameters, ESE Rules, and execution-controlling structures called Focus Control Blocks (IBM FSD's internal HiPER system -- for High Performance Embedded

Reasoning -- is our current approach to an inference engine for embedded KBS and would involve the generation of somewhat different rule syntax; cf. Arbabi & Highland, 1986, pp. 33-44)

While the few discussions of KBS life-cycles that are published certainly do open up a host of new opportunities for "smart" tools, this present life-cycle offers even more openings with its elaboration of the initial acquisition phase into the 3 paired cycles, prior to any prototype development.

KNOWLEDGE BASE VERIFICATION

The second phase of the expanded Knowledge Acquisition stage, shown in Figure 3, involves the semantic interpretation and coding of obtained knowledge into a formal system which is governed by a new evolving approach we call KNOWLEDGE REPRESENTATION THEORY (KRT). Not only does KRT provide opportunities for formal verification, as summarized here, but also it appears more and more likely that it can guide the development of structured interviewing techniques and the automation of subsequent knowledge transformation into KRT formalisms.

The basic KRT strategy is fourfold: (1) to develop a system of knowledge-elements which is rich and comprehensive enough to capture the complexities of modern-day systems and their development; (2) to develop a syntax and set of formalisms for expressing all of the semantics of a concept in an explicit, declarative manner, without recourse to deus-ex-machina tricks such as DEMONS or arbitrarily-inserted LISP (or other) code; (3) to generate a base of several hundred concepts for serving as a "core" for new customer-system knowledge (and for developing editing and other aids to facilitate elaboration of concepts); and (4) to perfect a set of procedures for automating formal verification of each new element of the knowledge-base.

Figure 4 illustrates a simplified view of KRT's basic knowledge elements: frames containing three types of slots ("c_" slots -- required header information; definitional slots, with no prefix; and "typ" slots -- slots indicating typical, but not necessary, information; only the first two are shown). One of the key architectural decisions of KRT was to require every slot and every slot-value to have its own separate definition frame. This provides the basis for a SEMANTIC-NETWORK-like richness of association among concepts in addition to the usual hierarchical inheritance and coupling structure provided by the generalization ("genzn") and specialization ("speczn") slots: one frame is linked to others not only by its "ISA" links (to use common, but objectionable, terminology), but also by every definitional or typical slot that occurs on it, and further by every value and every operator accompanying these slots (as shown by the arrows from the central "ADULT" frame to the other frames in Figure 4).

The very same architectural requirement which supports such associative richness also provides the basis for automated verification, in that a large number of syntax requirements have to be observed before all the elements on a single frame -- a single "knowledge element" -- can be said to be correct. Some of these requirements are listed in Figure 5, which also makes the point that, when all requirements are met for a single frame, then that frame can be termed a WELL-FORMED KNOWLEDGE ELEMENT, or WFKEL -- harking back to the concept of Well-Formed (logic) Formulas -- WFFs -- some decades ago.

Re-examination of Figure 4 indicates at least two problems for the ADULT concept-frame according to these well-formedness criteria: neither the operator GT nor the entity legal_adult have a separate defining frame. Were these two frames defined, and were the other syntax to be correct, then it would be the case that the ADULT frame would be a WFKEL, a verified trusted item of knowledge which will retain its lofty status no matter how used, as long as it and the frames on which it depends are not changed.

Verifiability, then, can be achieved by automated syntax-checking means given KRL and the frame formalisms (our specifications stay within the Horn-clause subset of first-order predicate calculus, and we do, in fact but not of necessity, use PROLOG to accomplish syntax-checking). While verifiability guarantees only syntactic correctness and overall reliability, still it greatly promotes validity and completeness, since the simple requirement of defining everything will force the elicitation of more and more information from the knowledge suppliers and will provide the best possible environment for them to discover for themselves that they forgot or mis-stated something.

REQUIREMENTS TRACEABILITY

This issue, in the large, deals with whether what you finally got is what you said you wanted in the first place; in the small it deals with seeing that this phase of development preserves the same system goals achieved in the previous phase. The process begins with the customer's expression of an "Operational Need", translated into a set of "requirements" which lead to an agreed-upon set of "specifications", often called the "A-Spec". This process may not be as much a traceability problem as it is a knowledge problem: often, particularly for problems involving the computerization of what people used to do, you don't really know exactly what it is you think you need.

It is the articulation of the A-Spec into a set of B-Spec design components, often called CPCIs (Computer Program Configuration Item) which is the biggest traceability difficulty. The CPCIs, the so-called B5s, are the result of taking the requirements as expressed in the A-Spec and -- via a series of complex architectural activities, trade-off studies, system engineering partitioning of hardware and software, and innumerable design

considerations -- producing a set of procedural prescriptions for accomplishing the previously declarative goals. The final steps are those of detailing each B5 into a much more explicit C5 design and then into its C5 implementation.

While the development of B5s from the A-Spec is surely the most challenging traceability problem, all steps are problematic because new information is introduced in each, primarily due to the increase in specificity (of candidate concepts), in interactivity of components, in quantification, and in the detail of resource consideration. Despite many years and many promising approaches, there is still no technique or product which can effectively assure that requirements from one stage to another not only can be traced but can be assessed as having been met.

While it is much too early to claim success for our KRT approach, we do believe it shows great promise for meeting these traceability goals. Whereas a major difficulty with traditional life-cycle traceability has been the fact that it is terribly difficult to compare the results of one stage with another -- they are so very different in structure and content -- the great strength of our technique is that the results of each stage will look extraordinarily similar. This is because, given Knowledge Representation Theory, they all are represented by the same frame formalisms and have considerable overlap in slot types and easily traceable elaboration of slot values.

Consider, for example, the invention of the wooden erase perncil in the early 20th centruy, say by the Acme Company. The initial Operational Need might have been simply to develop a cheap, easy-to-use, writing instrument whose marks were not to be permanent but somehow reversible. This situation is represented in the top panel of figure 6. Only a few of the needed slots are shown, but the key one is char_values, standing for "characteristic values", one of several means for formally representing ill-defined general performance criteria (the slot c_intent normally has the value "real", indicating that the concept describes real instances; c_link associates this ad hoc operational need for a pencil to the general well-developed base-concept of "operational_need"; c_next implies a frame sequence and specifies the next one in succession).

In the middle panel of Figure 6 the initial vague performance objectives have been sharpened up: the marks should not be permanent, the instrument should cost less than 10 cents, and "easy-to-use" now means "easy-to-hold" ans well as "easy-to-operate." In the bottom panel there is, for the first time, an expression of how the inst rument is to be constructed -- in the design frame -- in which the concept of "operation" is designed at this high level to be two modes, one for writing and one for erasing, with the "ease" criterion being asserted for the latter. The next frames might be detailed designs or the first implementations, in which case the name slot would not be c_name but i_name, which stands for "instance_name".

We believe that the two key processes of (1) tracing requirements and (2) assuring that they are met in each next phase is an automatable process if the specification procedures outlined in Figure 6 are followed and if each specification frame is assured to be a WFKEL, just like the basic concepts. This procedure can be used even if the implementation of a design was not formally generated from C5 descriptions, but rather from iterative rapid prototying, for example. All this is required is that the implementation be documented according to the KRT formalisms to produce frames, and these can then be compared to the Requirements and Design specifications.

SUMMARY

What has been presented here is admittedly an early program of research on KBS tools for working to retain all which is good about managing traditional software. The key thrust of the proposed techniques is to move as much as possible of the development of a KBS system to the initial life-cycle stage of Knowledge Acquistion, and then to accomplish very extensive verification by formal automated means. What is tantalizing about this kind of KBS emphasis is the possibility that some of those lofty ideals of traditional software development -- such as making changes in an operational system by changing the design and propagating that forward -- may actually have a greater chance of being satisfied for the new AI/KBS systems than for the traditional programming language ones.

REFERENCES

1. Arbabi, M. and Highland, F. D., "Embedding a fault diagnosis expert system into an existing C3 system". In Technical Directions, published by IBM Federal Systems Division, Vol. 12 (3-4), 1986, 30-34.

2. Miller, L. A., "The Knowledge Acquisition Approach to KBS Verification and Validation". Invited presentati on at a NASA conference on Validation and Verification of Knowledge-Based Systems, sponsored by the Information Sciences Division, Ames/Sunnyvale, California, 14 May 1987.

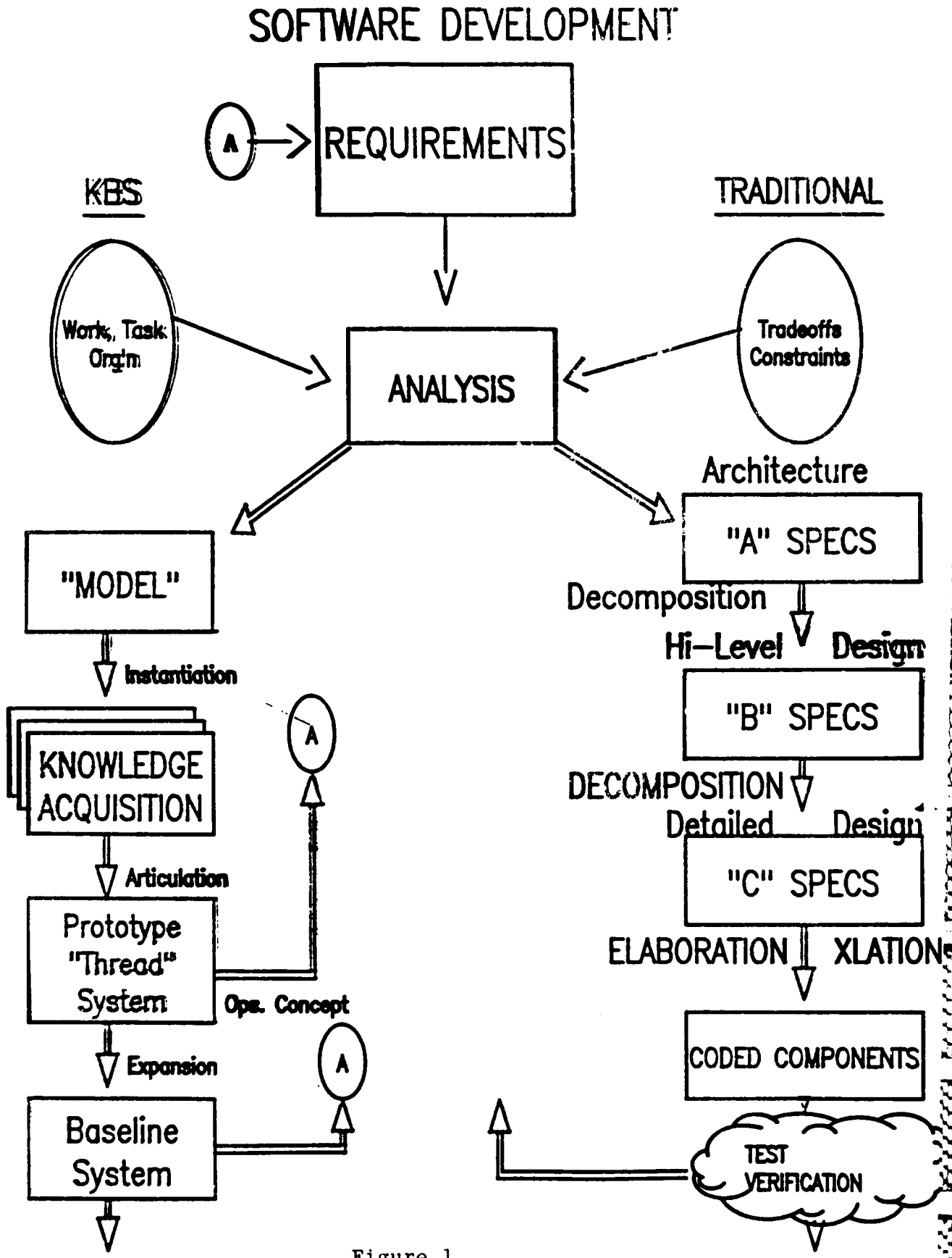

SOFTWARE DEVELOPMENT
A
REQUIREMENTS
KBS
TRADITIONAL
Work, Task Org'n
Tradeoffs Constraints
ANALYSIS
Architecture
"A" SPECS
"MODEL"
Decomposition
Hi-Level Design
Instantiation
"B" SPECS
A
KNOWLEDGE ACQUISITION
DECOMPOSITION
Detailed Design
Articulation
"C" SPECS
Prototype "Thread" System
Ops. Concept
ELABORATION
XLATION
A
Expansion
CODED COMPONENTS
Baseline System
TEST VERIFICATION
SD6

Figure 1

Life Cycles

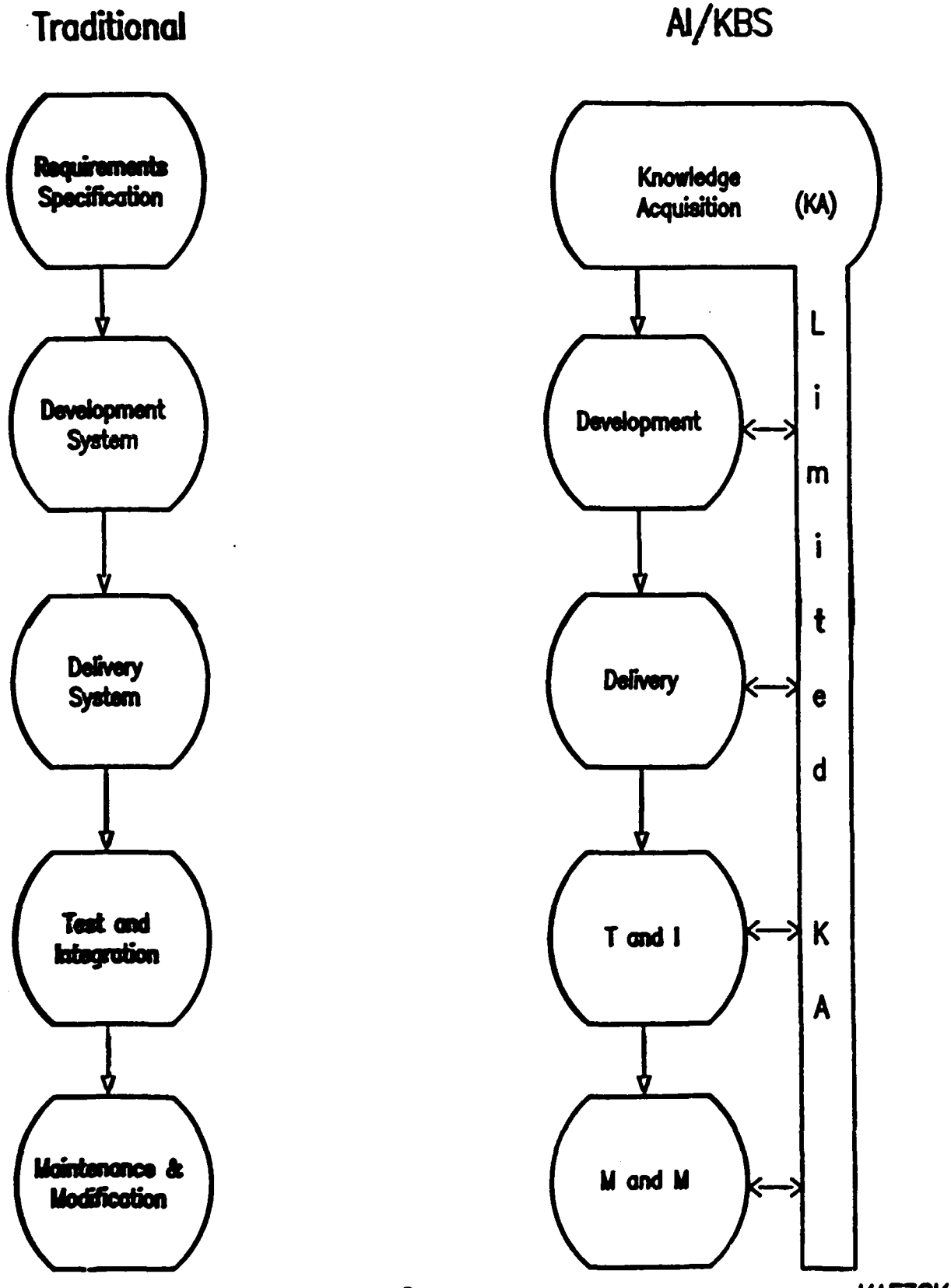


Figure 2

The New 3-Step KA Stage

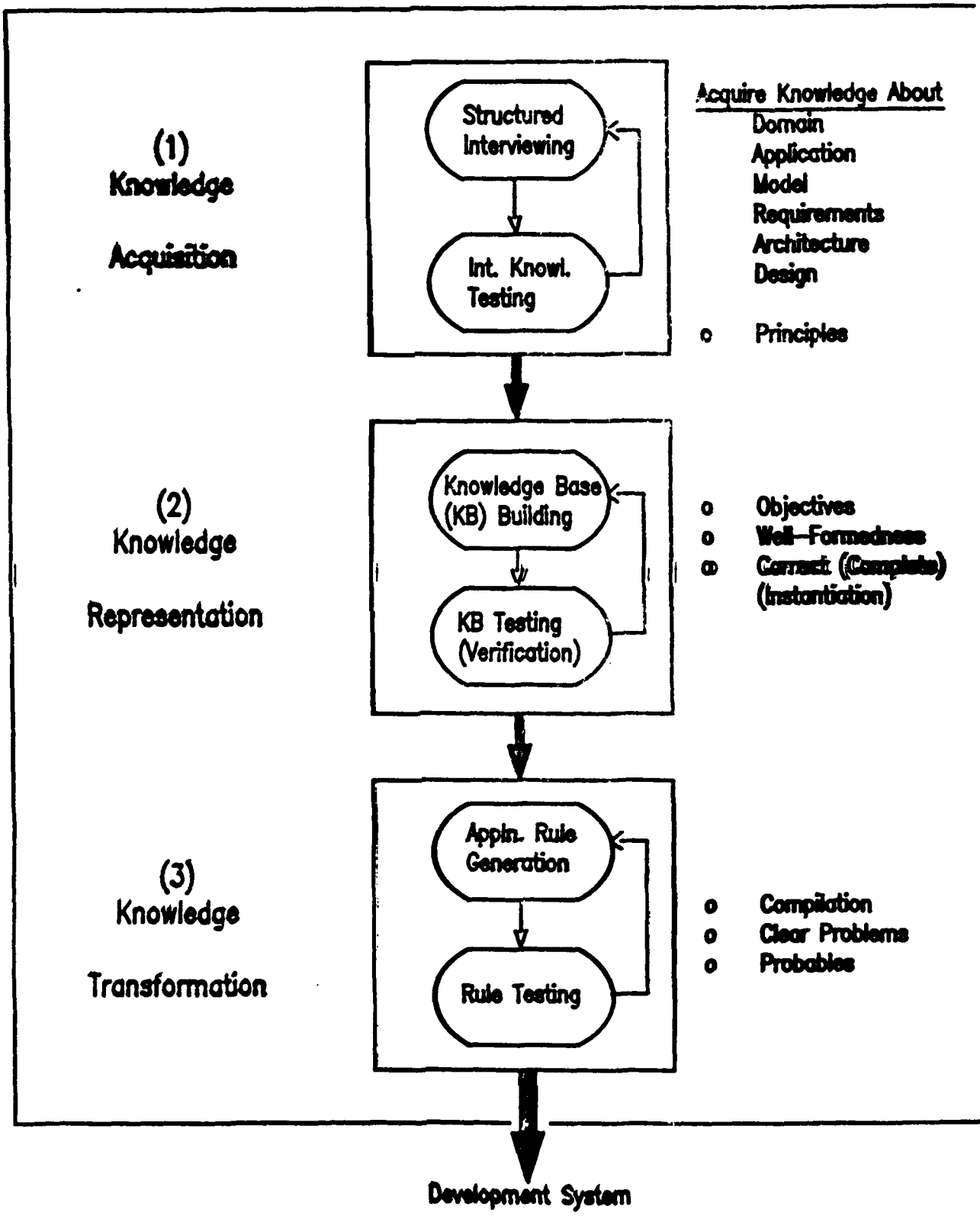


Figure 3

KAF10S

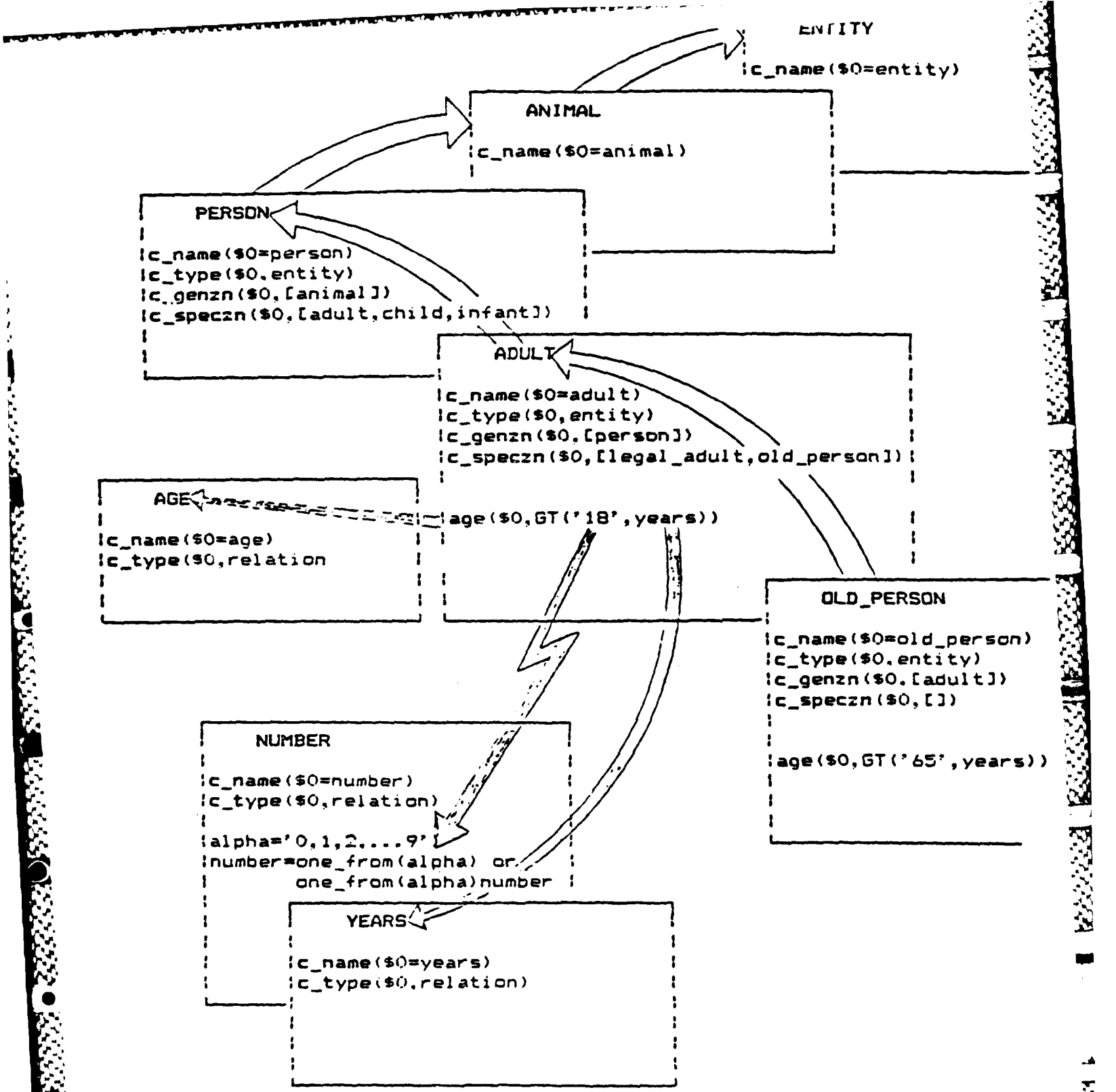
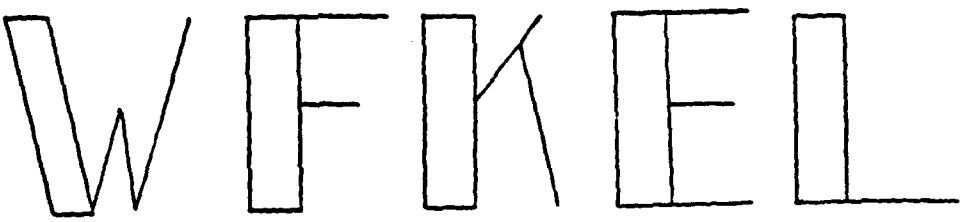

ENTITY
c_name($0=entity)
ANIMAL
c_name($0=animal)
PERSON
c_name($0=person)
c_type($0,entity)
c_genzn($0,[animal])
c_speczn($0,[adult,child,infant])
ADULT
c_name($0=adult)
c_type($0,entity)
c_genzn($0,[person])
c_speczn($0,[legal_adult,old_person])
AGE
c_name($0=age)
c_type($0,relation
age($0,GT('18',years))
OLD_PERSON
c_name($0=old_person)
c_type($0,entity)
c_genzn($0,[adult])
c_speczn($0,[])
age($0,GT('65',years))
NUMBER
c_name($0=number)
c_type($0,relation)
alpha='0,1,2,...9'
number=one_from(alpha) or
one_from(alpha)number
YEARS
c_name($0=years)
c_type($0,relation)

Figure 4

Knowledge Well-Formedness (WF) Criteria

- Does each Concept Element (Intention or Extension) have a Definition Frame?
- If an extension, is correct extensional syntax observed?
- Are Obligatory Slots present with correct syntax and values?
- Is there at least one definitional - slot?
- Does each slot have its own definition frame?
- Are the conventions of slot-value expression observed?
- Are the conventions of $VARIABLE usage observed?
- Does every value, operator, and argument have it own definitional frame?

THEN YOU HAVE A



(Well-Formed Knowledge ELement)

Figure 5

(A) OPERATIONAL NEED

```
c_name($0, writer.oper_need)
c_intent($0, goal(ACME))
c_link($0, operational_need)
c_next($0, writer.reqs)


primary_use($0, to_write(nil))
char_values($0,  cheap, not_permanent, easy_to_use )
```

(B) REQUIREMENTS

```
c_name($0, writer.reqs)
c_prev($0, writer.oper_need)
c_next($0, writer.design)
    .
    .
    .
primary_use($0, to_write(($A1 = role(actor, person)),
      role(instrument, $INSTR), role(medium, paper),
      consequences($P = on(marks, paper))))
permanence($P, not(permanent))
upper_bound(retail_cost($INSTR, $VAL, cents), LT($VAL, 10), cents))
difficulty(to_hold($A1, ($A2 = role(object, $INSTR)), low)
difficulty(to_operate($A1, $a2), low)
```

(C) DESIGN

```
c_name($0, writer.design)
c_prev($0, writer.reqs)
c_next($0, writer.instance)
    .
    .
    .
operations_modes($0,  to_write, to_erase )
    .
    .
    .
difficulty(to_erase($A1,$A2), low)
```

Figure 6

THE KNOWLEDGE INTEGRATION TOOL:

KNOWLEDGE INTERFACE

(A KNOWLEDGE BASED SYSTEM DEVELOPMENT ENVIRONMENT)

by

Philip H. Newcomb

Boeing Advanced Technology Center for Computer Sciences
Boeing Computer Services
P.O. Box 24346, MS 7L-64
Seattle, Washington 98124-0346
(206) 865-3431

The Knowledge Integration Tool: Knowledge Interface [1]

ABSTRACT

The Knowledge Integration Tool (KIT) is a Knowledge Based Environment for developing and using Knowledge Based Systems.

This paper contains excerpts from a thesis which studies the properties of both these system types and attempts their synthesis in the KIT, an advanced prototype of a Knowledge Based System Development Environment (KBSDE).

The KIT facilitates rapid construction of programming environments and end-user application software. Using the KIT, conceptual models can be created composed of conceptual primitives for agents, activities, causality, goals, methodologies, objects, resources, states, and time. A Knowledge Based Environment and the Knowledge Based Systems from which it is composed employ the conceptual model to engage in autonomous intelligent action to assist the user in his activities. The KIT employs user profiles to construct highly specialized environments tailored to focus the user's attention upon just the knowledge and operations pertinent to his activities.

The KIT possesses the rudiments of an architectual infrastructure which supports development and utilization of Knowledge Base Systems and Knowledge Based Environments within the Knowledge Based Corporation.

Key Words: Artificial Intelligence, Distributed Knowledge Base, Knowledge Based Corporation, Knowledge Based Design, Knowledge Based Environment, Knowledge Based Interface, Knowledge Based Project Management, Knowledge Based Programming, Knowledge Based Software Engineering. Knowledge Based System Development Environment,

---

[1] I would like to acknowledge the contributions of Dr. M. Fox and A. Sathi to the theoretical foundation of the Knowledge Integration Tool, and the helpful comments of Dr. B. Kadzierski, Dr. W. Tichy, Dr. A. Matsumoto, B. Ward, C. Gill and R. Speigle, my fellow researchers on the Crystal Project.

1. Introduction

The advent of high-end LISP machines and the availability of object oriented programming languages have made possible the class of knowledge based integrated tool environments. Such environments are characterized by the ease with which objects, the relationships between them, and higher level concepts composed of such objects and relationships can be represented graphically. Some such systems provide multiple views of knowledge presented based upon a model of the user and his application. Depending upon the sophistication of the user interface, the user may be able to control or change the way the system provides views of the knowledge base. Such systems usually provide automatic graphical draw facilities with capablites usually dependent upon the complexity of the graphs and the capabilities of the 'layout engine'. The user interface generally provides icon specific pop-up menus, interactive viewports, multiple display ports, and interaction with the user by means of high-resolution bit mapped displays, pointing devices, and key board inputs. [2] [3]

This paper provides an overview of the properties of the Knowledge Integration Tool, a 'Knowledge Based Systems Development Environment' which serves as a development platform and delivery vehicle for Knowledge Based Systems and Knowledge Based Environments (i.e. computerized environments constructed to manipulate knowledge bases with knowledge based systems). The functional and structural features of the Knowledge Interface, a fast and flexible interface for accessing knowledge and capabilities in the KIT, is described in some detail.

While the KIT may be used for program development and program construction, it can be effectively employed to gain access to, to reuse, to understand, and to modify already existing knowledge bases and knowledge based systems. The KIT inherits from its knowledge based foundation the ability to be rapidly extended to new problem domains in hours and days rather than in the months, or even years, customarily anticipated for conventional systems.

---

[2] Smith, R.G, Dinitz, R., Barth, P., "Impulse-86: A Substrate for Object- Oriented Interface Design," ACM, 1986, p. 167-175.

[3] Tichy and Ward, "A Knowledge-Based Graphical Editor," submitted to ACM Transaction on Graphics, special issue on user interfaces, Carnegie Group, Inc., July, 1986.

Properties of Knowledge Based Systems Development Environments

To help provide a focus for the discussions of the Knowledge Interface which follow, the capabilities of a Knowledge Based System Development Environment (KBSDE) will be listed below. A KBSDE should support many of the following features:

1. Incremental knowledge based construction: knowledge-of-the-small.

2. Knowlege based project management: knowledge-of-the-many.

3. Knowlege based management and control: knowledge-of-the-large.

4. A Knowledge Based approach: knowledge of the goals, task and domain of the process to which it is applied. This knowledge may range from highly specific models of the problem domain to which it is applied to more general knowledge about how to generate a user interface based upon a user or activity model.

5. A model of one or more (software engineering) methodologies. This knowledge may be integrated as domain knowledge into the knowledge base to be used within a model of user interaction.

6. An integrated rather than an incremental approach.

7. A sophisticated user interface with icon specific commands, pop-up menus, interactive viewports, multiple display ports, and interaction with the user by means of high-resolution bit mapped displays and automatically drawn displays.

8. Multiple views of underlying knowledge based upon a model of the user composed of, but not limited to, objects, relationships, and higher level concepts. The knowledge base may also possess knowledge of behavior components including rules, functions, demons, methods, productions, etc.... Such entities can be presented in tables, graphs, and textually, and the user should be able to control or change the way the system provides views of the knowledge base.

9. Facilities for rapid construction of extensions of the KBSDE to other problem domains.

The problem encompassed by these requirements is so very broad in scope that even if a single system can satisfy them there is a risk that it would be perceived as large, unwieldy, and difficult to use and understand. Therefore, substantial effort has been placed into developing an architectural structure for the KIT which will deliver the KIT's capabilities without overwhelming its users.

KIT Components Overview

The KIT's major components are described below:

o The *Knowledge Interface* is the KIT's top level knowledge interface and command shell. It is much like an encyclopedia of knowledge which provides access to all accessible knowledge bases and knowledge based systems.

o *Knowledge-Of-The-Small* is concerned with helping an individual programmer or designer develop a single program in relative isolation from other programmers.

o *Knowledge-Of-The-Many* is concerned with helping groups of managers, designers, programmers and users manage, develop and use software resources in coordination with each other.

o *Knowledge-Of-The-Large* is concerned with controlling the evolution of the knowledge base and keeping it in a well defined state.

o *Knowledge-Of-The-Moment* is concerned with focusing attention upon knowledge and capabilities which are pertinent to the user's choice of activities, roles and goals. †

Knowledge-Of-Itself provides the user with a control panel which permits the user to instantly reconfigure the knowledge based environment based upon his profile and his needs.

The KIT's major subcomponents attempt to satisfy the following requirements of a KBSDE:

1. The Knowledge Interface (human and knowledge interface) supports 4, 5, 6, 7, 8.

2. Knowledge-Of-The-Small (knowledge based system design and construction) supports requirements 1, 6, 7, 8, 9.

3. Knowledge-Of-The-Many (project management and knowledge based environment construction) supports requirements 2, 4, 6, 7, 8, 9.

4. Knowledge-Of-The-Large (distributed knowledge base management and configuration control) supports requirements 3, 6, 7.

At this writing, the KIT is maturing as a delivery vehicle for knowledge based technology and is already a dynamically redefineable environment which can alter itself to the needs of its users. As a member of the class of knowledge based integrated tool environments, the KIT Knowledge Interface acts as a flexible, general purpose interface for a wide class of users to many kinds of applications and knowledge. The KIT's generality comes from its ability to use a profile of a user and knowledge context and command context specific constraints to provide access to just the tools and knowledge which a user needs to do his job. The KIT's flexibility comes from the ease with which the KIT can be extended to incorporate new application environments as they are developed, even if they were not originally developed with or as part of the KIT.

† A subset of this capability: *Knowledge-of-its-History* appears in implementations of KIT as of June 1986.

2. Capabilities

The Knowledge Interface serves the Knowledge Integration Tool as a variable width communication channel with the Knowledge Based Environment (KBE) and acts as an open ended framework for architectural extension. The rest of this paper reviews the way the Knowledge Interface can be used to partition the knowledge and command structures of the knowledge based environment.

2.1. Large Grain Knowledge Partitions

Knowledge based systems of any sophistication tend to become large and complex. A knowledge based environment may be composed of dozens of different kinds of knowledge based systems. The KIT Knowledge Interface has been constructed to reduce the environment's complexity while at the same time enhancing the user's ability to fully understand and use it. A prominent, immediately noticeable feature of the KIT is a command window across the top of the screen where there are icon commands for switching between knowledge partitions. Selection of a knowledge partition determines the kind of knowledge with which the user interacts. A user can change knowledge partitions instantly by *simply selecting one of the labeled partitions.* Research at Carnegie Mellon University on the Gandalf project identified three kinds of knowledge which should be supported by any software development environment: programming in the many, the large and the small. [4] The KIT emphasis is placed upon knowledge based environments rather than software development environments (SDE). Many of the same kinds of issues are present in a KBE as in a SDE but the domains are quite different. While a SDE is concerned with managing resources relative to programming projects, a KBE is concerned with managing all knowledge based resources which includes all knowledge based applications and their knowledge bases as well as programming projects. The KIT Knowledge Interface has three top buttons labeled Knowledge-Of-The-Small, Knowledge-Of-The-Large, and Knowledge-Of-The-Many shown in Figure 1. Choice of any one of these buttons results in display on the screen of an icon command window which provides a detailed partitioning of knowledge within each of the top level partitions -- to be

[4] Habermann, A. N., Notkin, D.S., "The Gandalf Software Development Environment," Carnegie Mellon University, Pittsburg, PA, January, 1982, p. 1.

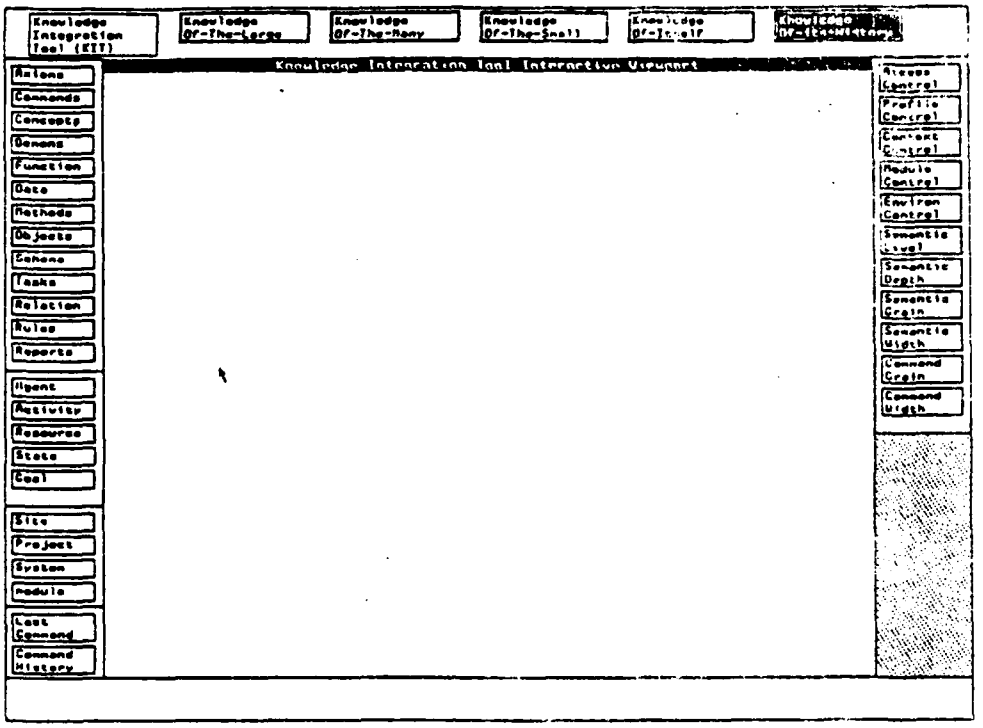


Figure 1: The KIT Knowledge Interface

discussed in a moment. In addition to these three broad knowledge partitions, two additional selections are provided: Knowledge-Of-The-Moment and Knowledge-Of-Itself. Knowledge-Of-The-Moment, a term coined by the author, provides a dynamically constructed knowledge based environment which varies with a user's choice of activities, role or goals. Knowledge-Of-Itself provides a control panel which lets the user instantly reconfigure the knowledge based environment or change his profile.

2.2. Small Grain Knowledge Partitions

Selection of any one of Knowledge-Of-The-Small, Knowledge-Of-The-Large, and Knowledge-Of-The-Many knowledge partitions results in display of command windows on the screen with icon commands for selecting even finer subdomains of knowledge. Selection of one of the labeled icon commands on the command window results in display upon a pop-up menu of entities within the associated subdomain. This *Entity Display* will vary depending upon the current environmental control

settings. The user can either browse deep within a knowledge taxonomy or select from a *Command Display* commands to perform operations upon one or more entities chosen from the Entity Display. The process of browsing is initiated from one of the top level entities of a Small Grain Knowledge Partition. The user can browse through knowledge by choosing one or more entities from the Entity Display and then selecting the 'DESCEND' menu choice. This choice results in presentation upon another Entity Display menu of a set of entities related to the previous entities to a specified depth and width. The browsing process can continue as long as the user wishes. When the user finds upon an Entity Display one or more entities he is interested in, he can select the entities, and select the 'DONE SELECTING' menu choice. A selection of entities from an Entity Display determines a ' *Knowledge Context*. Once a Knowledge Context is chosen, the Command Display will appear. Commands shown at any time upon a Command Display are usually specific to the type of entity selected and the context of its use. After selecting one or more commands from the Command Display the commmands are performed in the sequence chosen upon the selected entities. Often the command action results in invocation of a subenvironment of the knowledge based environment.

2.3. Entity Displays

The 'Entity Display' shown in Figure 2 is a dynamically generated pop-up menu which displays one or more types of entities. An Entity Display is much like a carefully organized index or the table of contents of an encyclopedia which provides structured access to knowledge. A semantically sound knowledge base is usually organized within taxonomic hierarchies. The Entity Display employs the relations used by epistemological structuring mechanisms of the knowledge base for browsing through knowledge. Knowledge is organized into networks connected by special epistemological level structuring relations, such as: instance, is-a, has-component, has-part, subset, member-of, etc.... The labelled entities at the level of Small Grain Knowledge Partitions are generally root entities of very wide and deep entity classes. Membership in an entity class is defined through a relatively small set of epistemological level relations which connect all accessible schemata in an arbitrarily large and deep network.

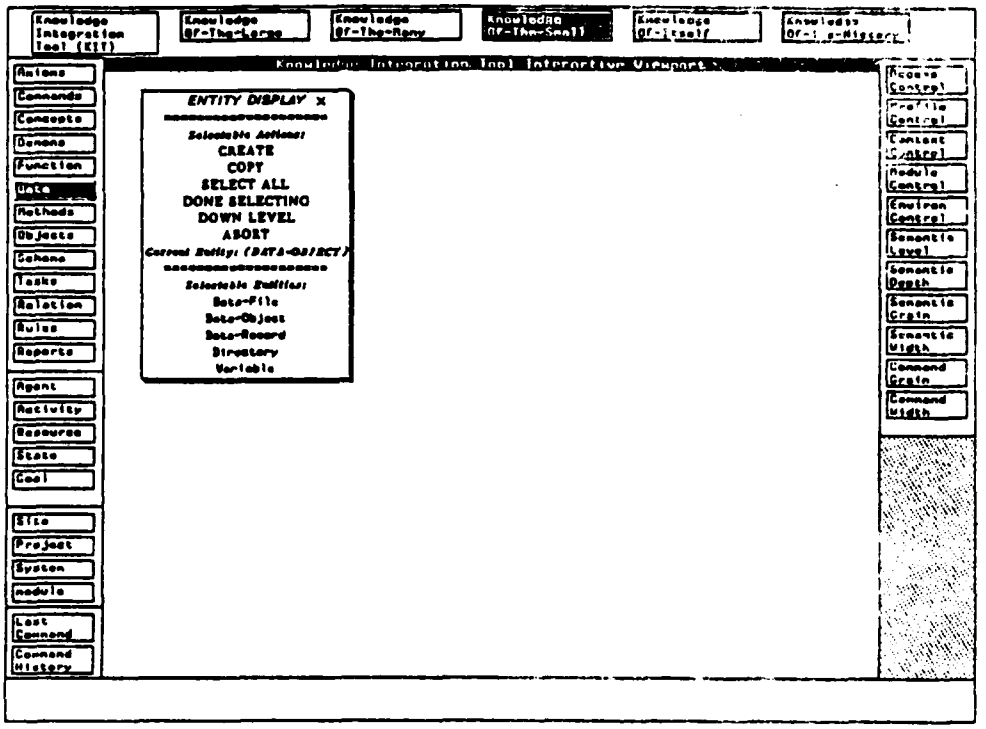


Figure 2: A Sample Entity Display

Every entity created in the KIT is a schema and is integrated into some existing network, which means it is -- or should be -- introduced as an individuation, instance or manifestation of some other entity.

2.4. Command Displays

The 'Command Display' shown in Figure 3 is a dynamically generated pop-up menu which displays one or more types of commands. A Command Display is much like a sophisticated command and control center which provides dynamically structured access to the capabilities of a complicated machine. Many software systems -- notably most operating systems -- provide little help to the operators who have to make choices about how to use the software system's resources. An operator is required to make dozens of decisions about how to use a system's resources. Decision making within a complex environment requires knowledge of the kinds of choices possible within a particular context.

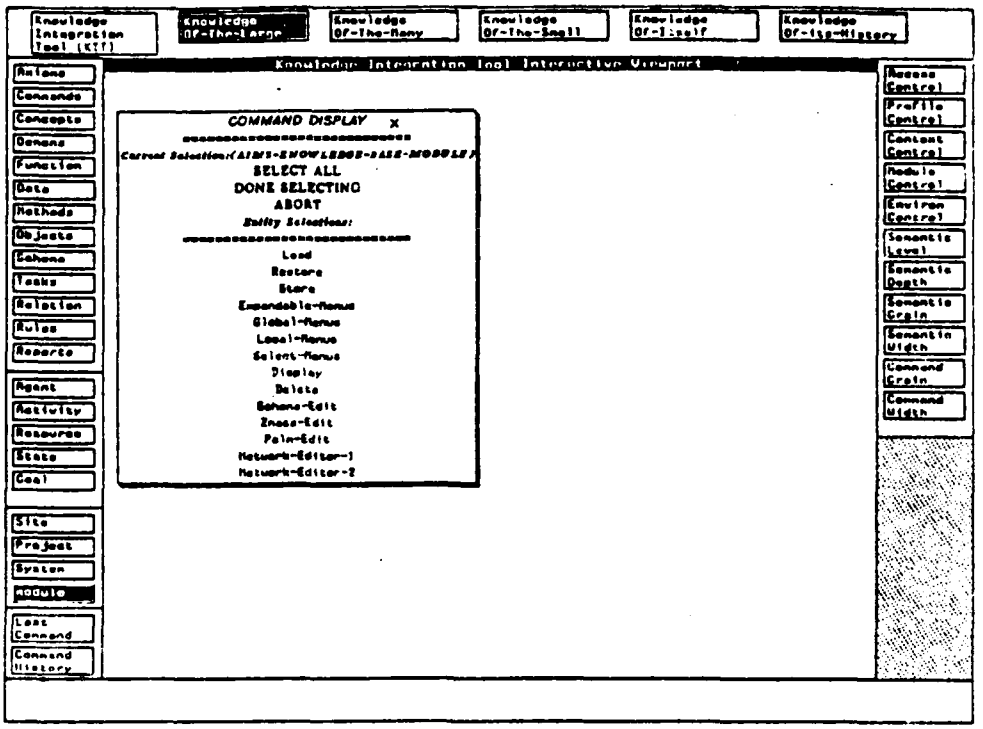


Figure 3: A Sample Command Display

It is possible to create commands and command selections which are useful and specific to a particular context. In fact, most commands are meaningless out of context, and most commands only exist and make sense within a specific context. The context of a decision within the KIT is determined by a profile of the user's role, the user's selection of environmental control settings, and the Knowledge Context. These factors makeup a set of constraints. The Command Display menu is a dynamically generate set of commands which satisfies the set of constraints. The user's selection of one or more commands from a Command Display defines a *Command Context*. Taken together large and small grain knowledge partitionings and the fine grain discrimintation provided by of Knowledge Contexts and Command Contexts reduce the complexities of the environment sufficiently for a user to perform complex problem solving tasks intelligently with relatively little or no training.

2.5. Command & Entity queues

Any number of entities and commands can be queued for sequential execution. Most commands expect to be passed an entity as an argument (actually functions associated with each command are passed the selected entity). If desired, all entities or all commands can be selected at once by choosing the 'SELECT ALL' menu selection. The ability to queue actions in this way improves the utilization of the user's time and helps him to focus his attention closely upon tasks at hand. The Knowledge Interface attempts to minimize keyboard inputs whenever possible by maximizing mouse selectable choices. It is rarely necessary for the user to type in the name of an entity -- except for when he first wishes to create it.

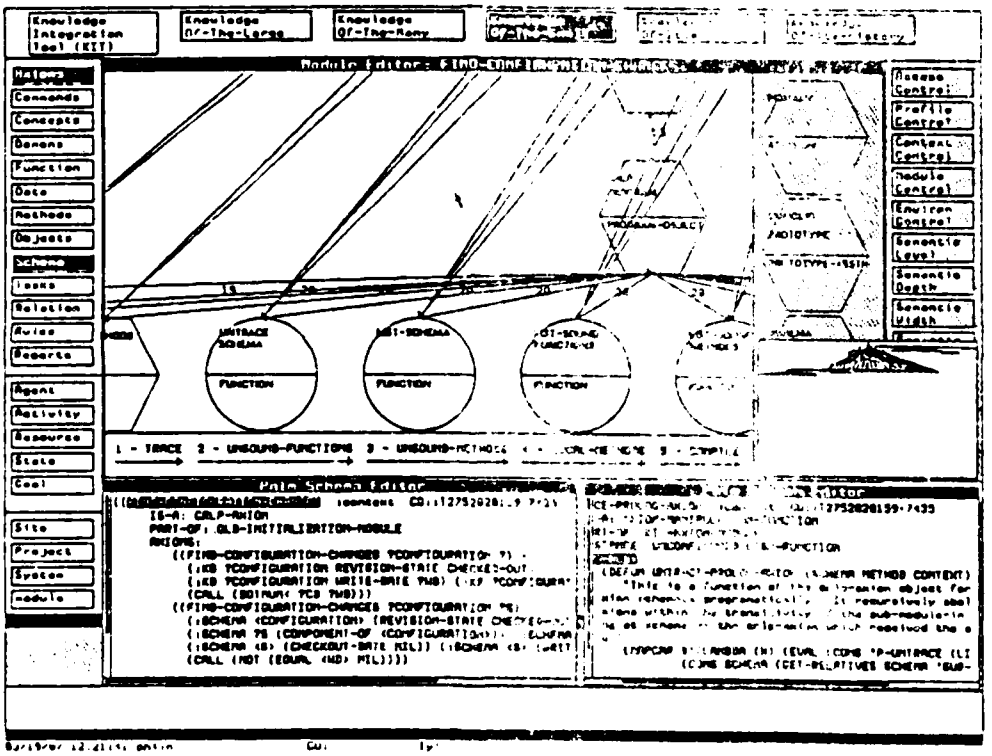


Figure 4: Multiply Queued Command Environments

Figure 4 illustrates how a sequence of commands can be used to invoked multiple simultaneously invoked command environments. In the diagram only one of the tasks environments is running while the others are paused. Of course, in addition to separate command environment invocation, the inter-

face can efficiently perform much more simple tasks, such as deleting displaying or individuating whole sets of selected entities.

2.6. A Knowledge Based Integrated Tool Environment

The KIT Knowledge Interface acts as a flexible, general purpose interface for a wide class of users to many kinds applications and knowledge. The user interacts with the KIT through high-resolution bit mapped displays using a mouse pointing device for command or object selection, and a keyboard for input of text and control sequences. The KIT employs icon commands upon icon command windows for many of its command selections. Several forms of pop-up windows and pop-up menus are used for displaying schemata and command selection. The knowledge base can be viewed and manipulated in a variety of graphical and textual ways provided by command environments. The user is able to freely move from one command environment to another by mouse movements and input sequences. The KIT user interface is written in CRL Command and Window System functions. The KIT's network graphics are generated by the Knowledge Craft (TM) Palm Network Editor, the Crystal Knowledge Based Graphical Editor or the Module Development Editor.

Palm Network Editor

The Palm Network Editor is suitable for displaying automatically drawn 'trees' of schemata and relations.

The Knowledge Based Graphical Editor

The KBGE can display heuristically drawn 'networks' of schemata and relations.

Module Development Editor

The MDE can display heuristically drawn networks of functions, schema, relations, and slots. The MDE uses the KBGE for its command structure and as a graphical layout engine, but it uses a powerful set of heuristics to select the choice of entities and properties it displays. It might be considered a '2nd Generation' KGBE.

The Knowledge Craft Coconut Schema Editor and the Palm Schema Editor are used for most manipulations of schemata.

The Coconut Editor

The Coconut Editor provides control over prompts on slots and input values to a schema. It can control the sequence in which schema slots are filled and the choice of slots to be displayed or edited

The Palm Schema Editor

The Palm Schema Editor permits nearly any update or edit -- even of meta information -- to be made directly to a schema within a Zmacs like editing environment. It does not, however, permit the same kind of value formating and verification control as is provided by Coconut.

Any number of interfaces are supported by the KIT. Interactive Display Windows are provided for long dialogues with the user. Interactive pop-up input windows are provided for the user to respond to input requests for one or more lines of text, and message displays are provided to display one or more lines of text to the user. The CRL-OPS (TM) and CRL-Prolog (TM) Workbenches are also accessible as tasks callable from within the KIT.

3. Knowledge-Of-Itself Interface

3.1. Environmental Controls

Selecting the Knowledge-Of-Itself subenvironment results in display upon a icon command window of a set of icons for controlling the way knowledge is presented to the user. Using these controls the user can adjust the 'depth', 'width' 'granularity', 'context' and 'semantic level' of knowledge and commands. These environmental parameters can be used to alter the user's interface with the KIT environment by increasing or decreasing the complexity and size of the KIT's command and entity displays. Figure 5 contains a sample view of panels which provide global control over the Knowledge Integration Tool's behavior.

Semantic Levels

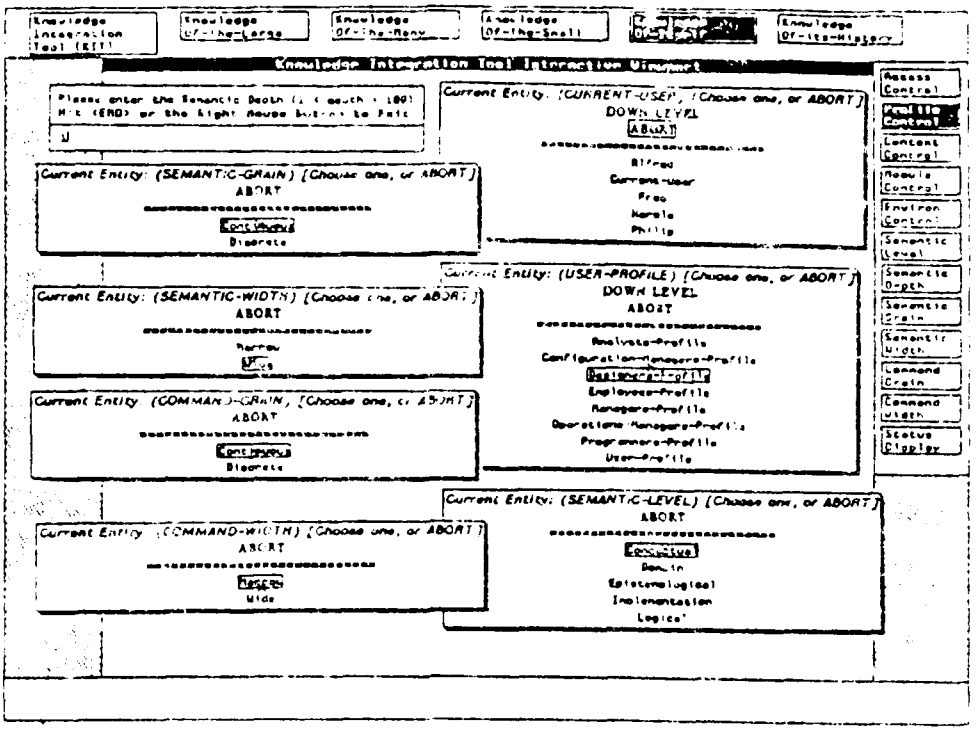


Figure 5: Knowledge-Of-Itself Interface

KIT knowledge bases are structured within a formally define epistemological structure. The KIT reflects this epistemological structuring of knowledge by permitting the user to choose the 'semantic level' at which he wishes to do his work. The semantic level is one factor in the decision the KIT makes about the kind of mechanisms provided to the user for manipulating the knowledge base. Entity and command displays will vary in content depending upon how the semantic level is set. The implementation, logical, epistemological conceptual, and domain semantic levels are currently supported by the KIT. Additional levels may need to be created to add even finer levels of knowledge discrimination.

Semantic Depth

The taxonomic structure of a knowledge base is defined by a semantic network whose depth is expressed as an integer number which describes the number of times a taxonomic relation is stepped

across in traversal of the network. The user can control the number of levels of the semantic network from which entities are taken for Entity Displays by setting the semantic depth.

Semantic Width

The Semantic Width controls the manner of display, whether the display mirrors the structuring of knowledge or collapses all knowledge into a single display. To display all reachable entities at once the Semantic Width can be set to wide. To display entities such that the taxonomic structure between them is preserved the Semantic Width should be set to narrow.

Semantic Granularity

Semantic Granularity controls whether semantic level barriers are crossed in the display of knowledge. When the Semantic Granularity is set to 'Continuous' smooth transition is provided across semantic levels, and the user is able to view knowledge at several semantic levels simultaneously. When set to Discrete only knowledge of a particular level is displayed. A typical application user, who usually functions at the domain level or higher, has no need to cross epistemological barriers. Knowledge Based System Designers, on the other hand, need knowledge of the more intricate semantic structurings of the knowledge base and require a means to follow smooth transitions over semantic level barriers.

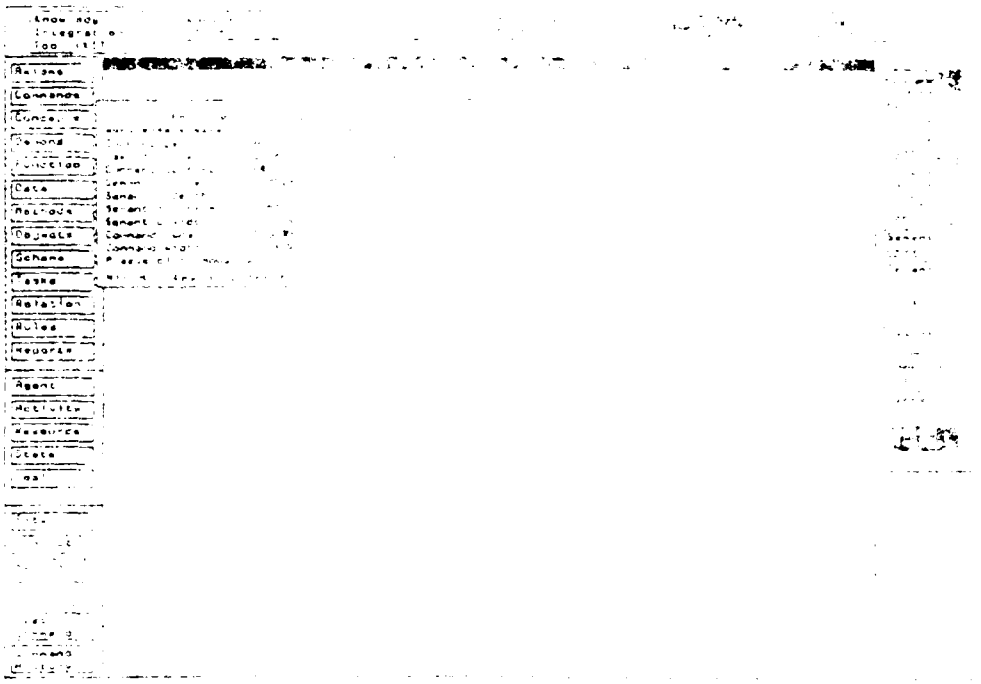
Command Granularity

Command Granularity controls whether semantic level barriers are crossed in the command displays. Continuous or Discrete settings are provided. When the Command Granularity is set to 'Continuous' a smooth transition is provided across commands of several semantic levels. When set to Discrete only commands of a particular semantic level are displayed.

3.2. Environmental Status

The Knowledge-Of-Itself command environment provides a display of the settings of environmental controls, the user's profile, and knowledge about the knowledge based environment's configuration upon the Environmental Status Display. This display shows the current state of the Knowledge Based

Environment [illegible]



4. Knowledge-Of-The-[illegible]

4.1. Browsing through [illegible]

It is essential [illegible] enable with the knowledge [illegible] Tool makes available [illegible] interface [illegible] Function [illegible] view of [illegible] Of-The-[illegible] clopedia of [illegible]

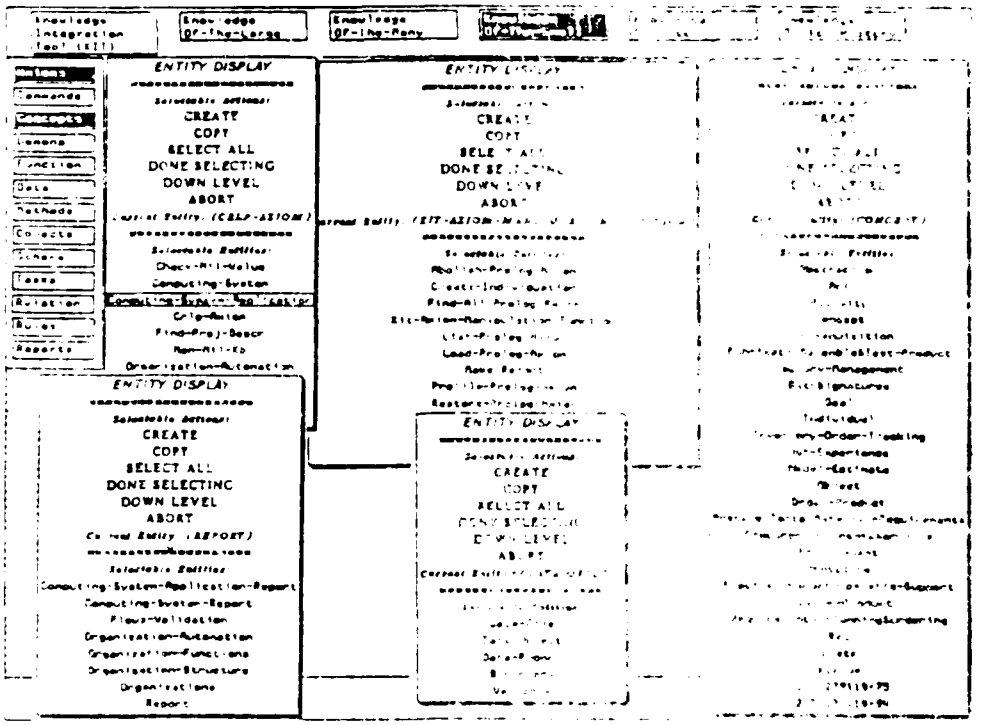


Figure 7: Knowledge-Of-The-Small Entity Display

partition is that the only programming entities which are [illegible] those which are in the workstation's virtual memory.

4.2. Selection of Programming Commands

Once a programming entity has been selected a command menu for manipulating the entity is provided which is appropriate for the chosen entity. Figure 8 shows sample command selections available through the Knowledge-Of-The-Small Command Display.

5. Knowledge-Of-The-Many Interface

5.1. Browsing through Activity Models

It is important for a knowledge based environment to serve many kinds of users and for users to be able to use software resources in coordination with each other. The Knowledge Interface provides

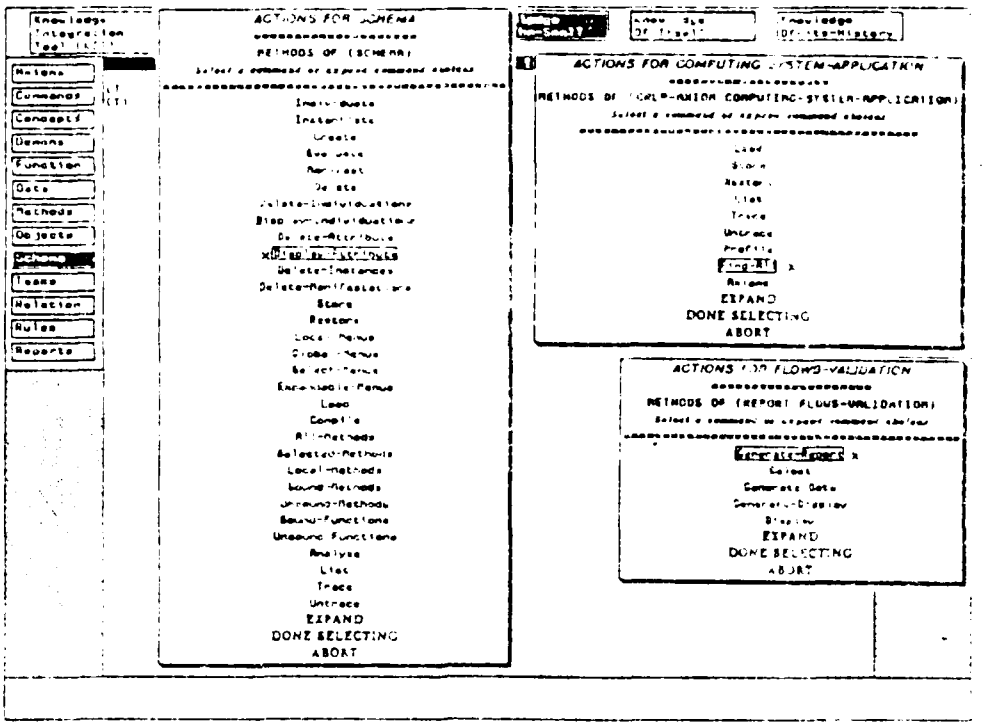


Figure 8: Knowledge-Of-The-Small Command Display

users with easy access to highly specialized application environments through the knowledge partition for Knowledge-Of-The-Many. This partition possesses a representation for project activities. Depending upon the semantic level of interaction which a user chooses, he may either design an activity representation by creating a model for it at the conceptual level, or he may develop a domain level application within constraints created by a conceptual model. During the construction of a conceptual model, a model developer may frequently switch from one semantic level to another in order to alter various characteristics of the environment being designed. The Small Grain Knowledge Partition of this interface holds knowledge of Activities, Agents, Objects, Resources, States and Goals. Figure 9 shows a view of some of the top level choices of entities accessible from within the Knowledge-Of-The-Many's Small Grain Knowledge Partition.

5.2. Selection of Activity Commands

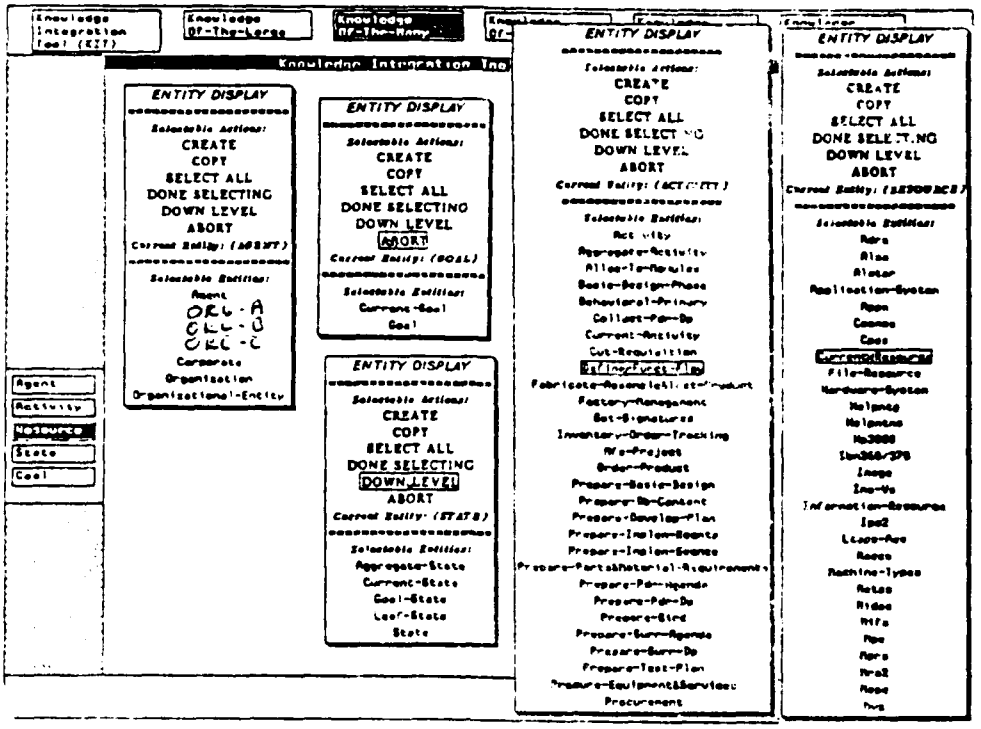


Figure 9: Knowledge-Of-The-Many Entity Display

Once the choice of an entity has been made a selection of commands for manipulating the entity should be provided which is appropriate to the kind of entity chosen, and the current environmental control settings. A manager, for instance, is likely to be concerned with checking on the progress of an activity or specifying the order of activities, so commands are provided which permit managers to review the PERT chart of an activity network or to modify a network which shows a sequence of activities. An activity designer is more like to be concerned with specifying relationships between prototypical activities and resources used in creating or transforming objects, so commands exist to provide him with specialized environments for designing various kinds of activity networks.

Figure 10 shows the command selections available through the Knowledge-Of-The-Many Command Display

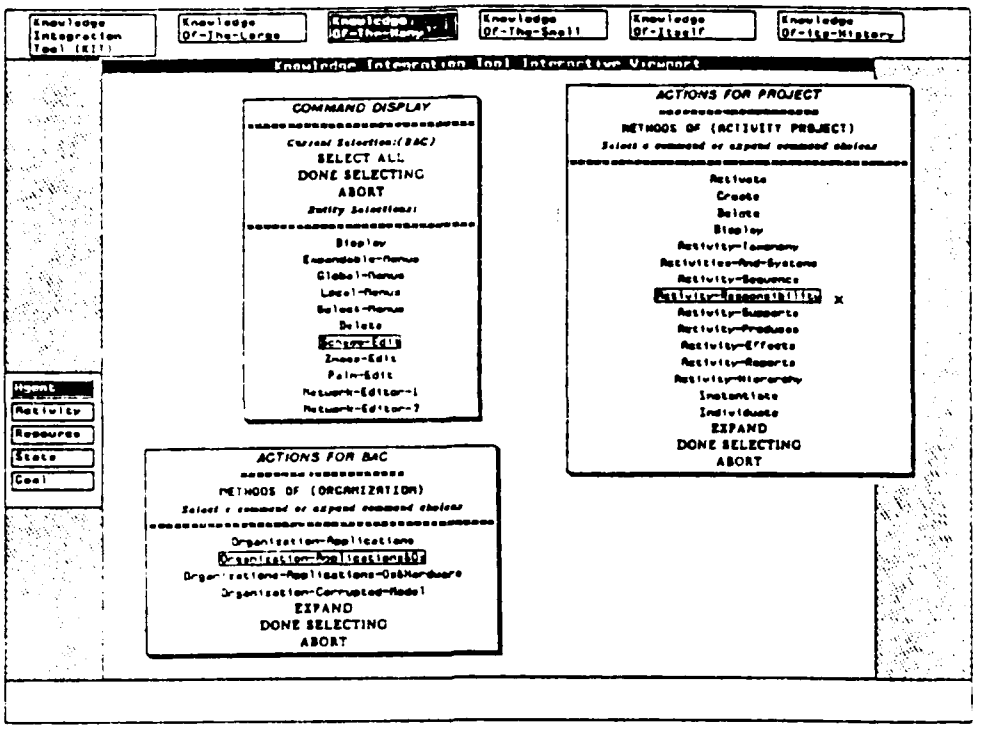


Figure 10: Knowledge-Of-The-Many Command Display

6. Knowledge-Of-The-Large Interface

6.1. Browsing through Configurations

It should be possible for an authorized user to obtain access to corporate information resources regardless of how the knowledge is distributed. The Knowledge Integration Tool provides access to a library of information resources through the Knowledge-Of-The-Large interface. The Small Grain Knowledge Partition of this interface holds a library of Sites, Projects, System, and Components from which the user's software configuration can be constructed. Figure 11 The diagram below shows a view of some of the top level choices of configuration components accessible from within the Knowledge-Of-The-Large's Small Grain Knowledge Partition.

6.2. Selection of Configuration Commands

Once a choice of configuration components has been determined commands for cataloguing,

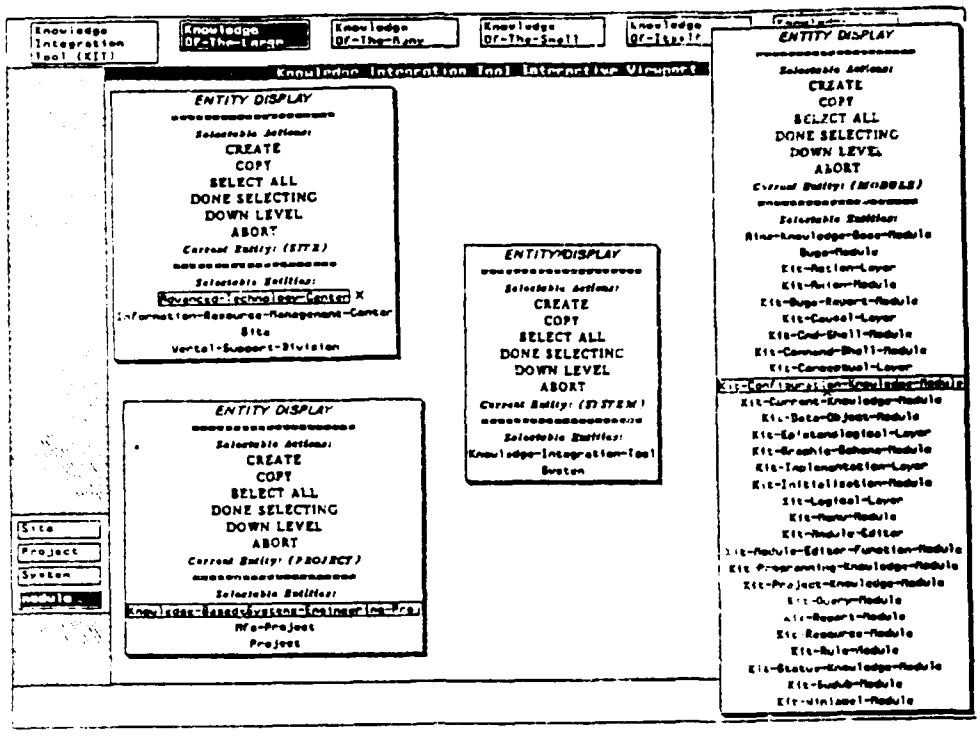


Figure 11: Knowledge-Of-The-Large Entity Display

locating, checking-in and checking-out, creating baselines, and so on, is provided. Figure 12 shows the command selections available through the Knowledge-Of-The-Large Command Display.

7. Knowledge-Of-The-Moment Interface

7.1. Browsing through Entities of the Moment

A final degree of environment specialization is provided which directs the attention of the user to a network of closely related entities. While the Knowledge-Of-The-Many interface provides users with access to a wide selection of specialized application environments, the Knowledge-Of-The-Moment interface permits the scope of knowledge to be narrowed. For instance, it is able to identify just the resources and objects, etc... which are related to a single activity. The knowledge partition defined by the Knowledge-Of-The-Moment interface defines a cross section of semantically interrelated schemata and the commands suitable for their manipulation. This partition may be defined at any semantic level.

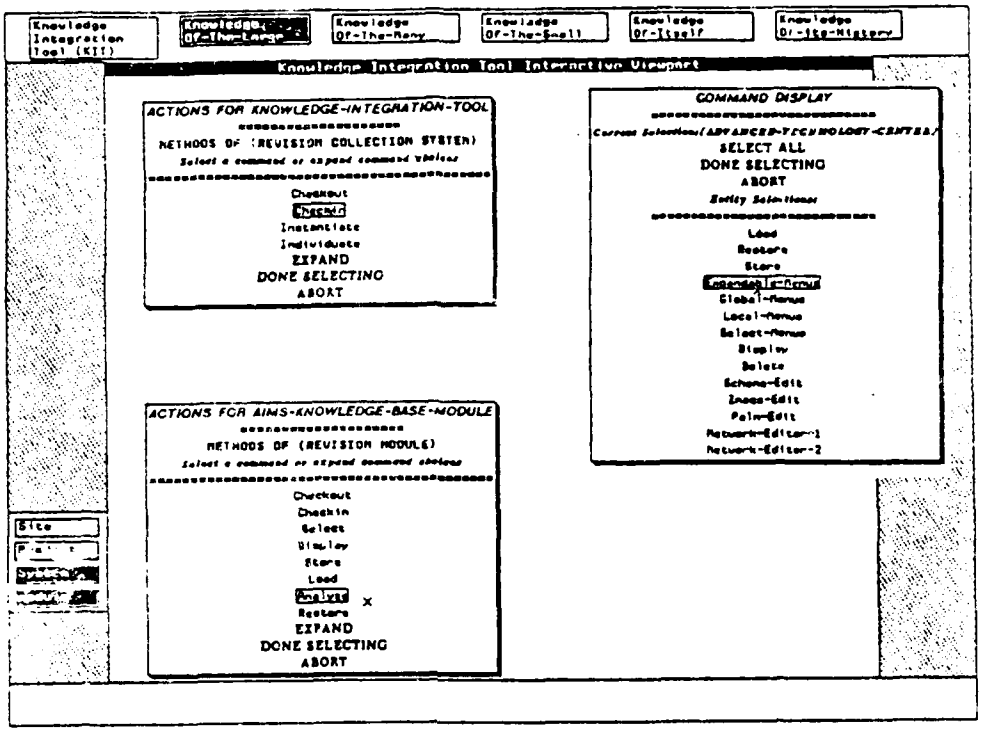


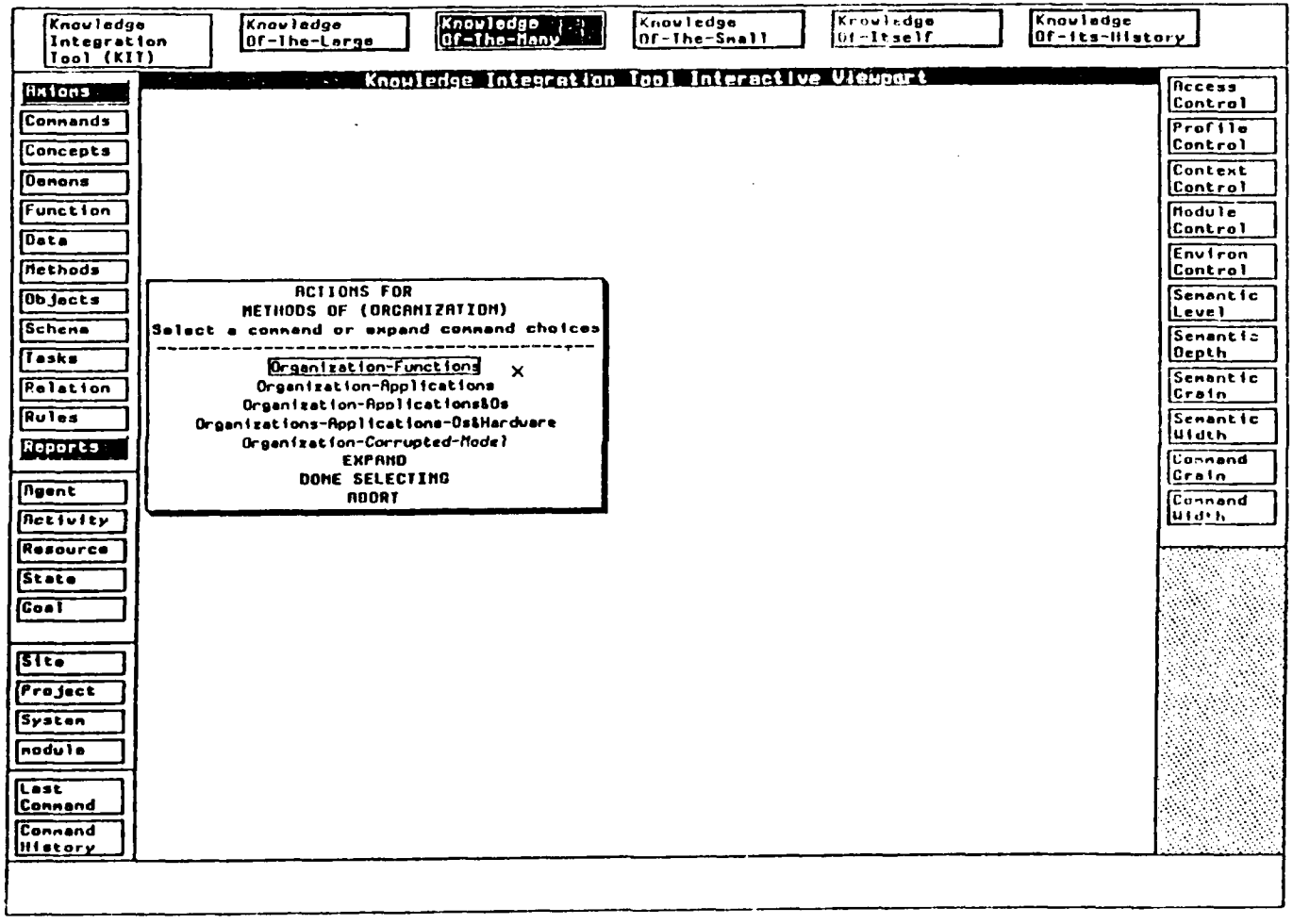
Figure 12: Knowledge-Of-The-Large Command Display

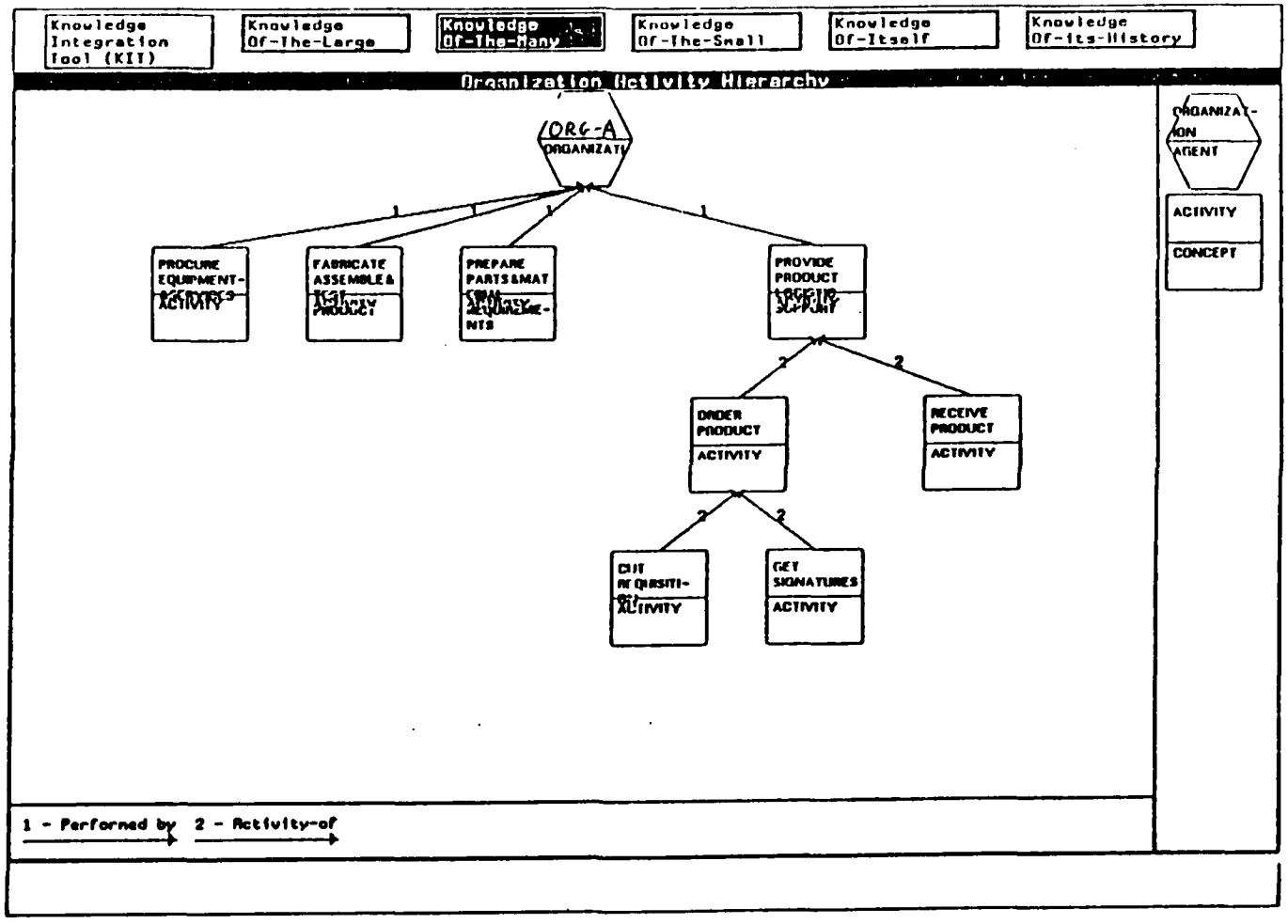
The dimension of the partition is determined by natural boundaries within the modeled knowledge, by definitions attached as meta-knowledge to certain objects, or by chronicling of information about the history of user knowledge and command context choices (i.e. recent Entity Displays and Command Displays).

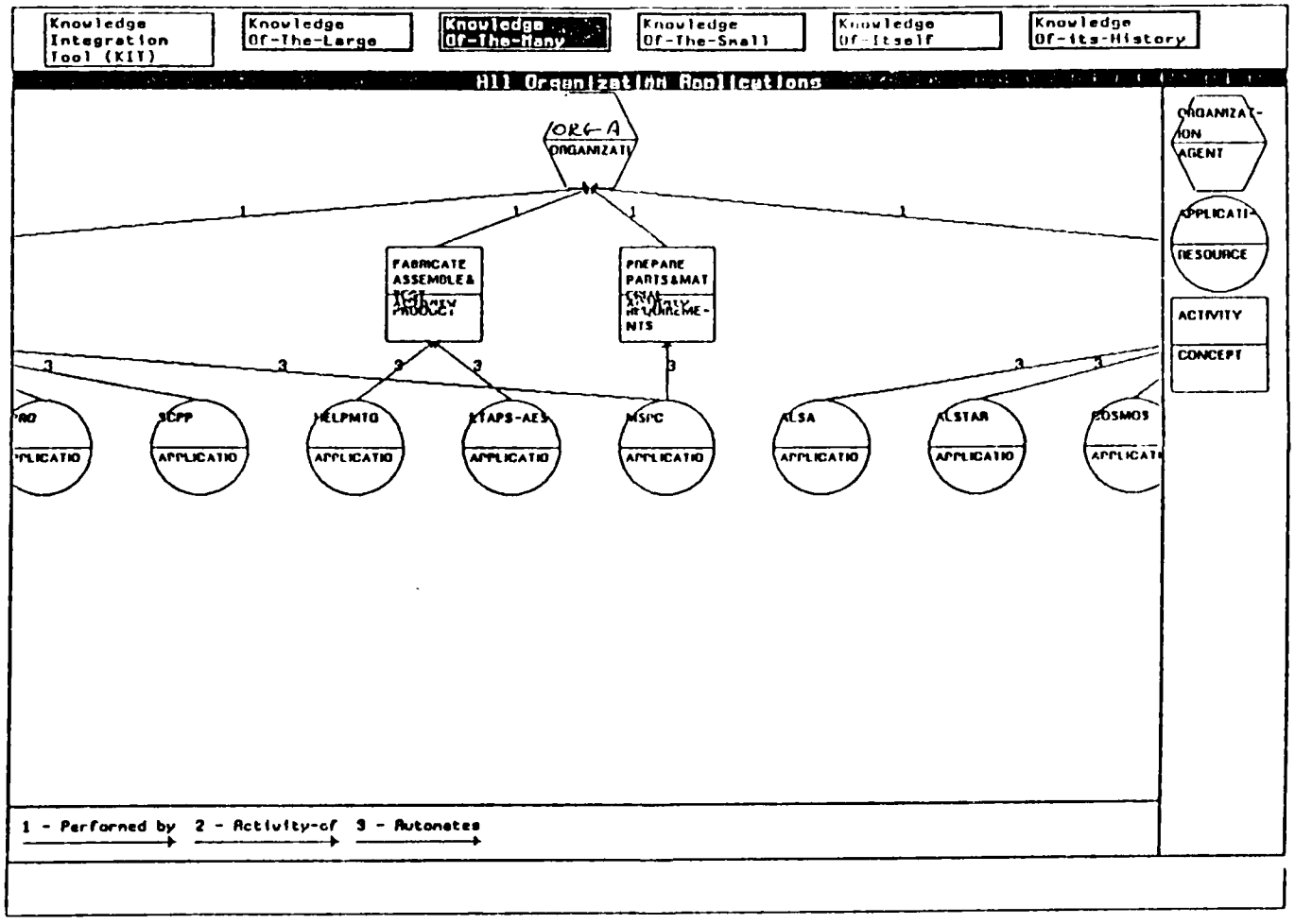
The Knowledge Interface provides powerful mechanisms for effectively utilizing resources in the knowledge based environment. Motivation for the construction of this interface was prompted in part by analysis of the Carnegie Group's Knowledge Craft Interfaces by students in the Advanced Technology Center/Knowledge Craft course. It is the author's hope that this interface will help Knowledge Craft users to utilize Knowlege Craft more effectively. Screen samples of a prototypical knowledge based application for automation planning, which was rapidly created with the Knowledge Integration Tool, are included in Appendix A.

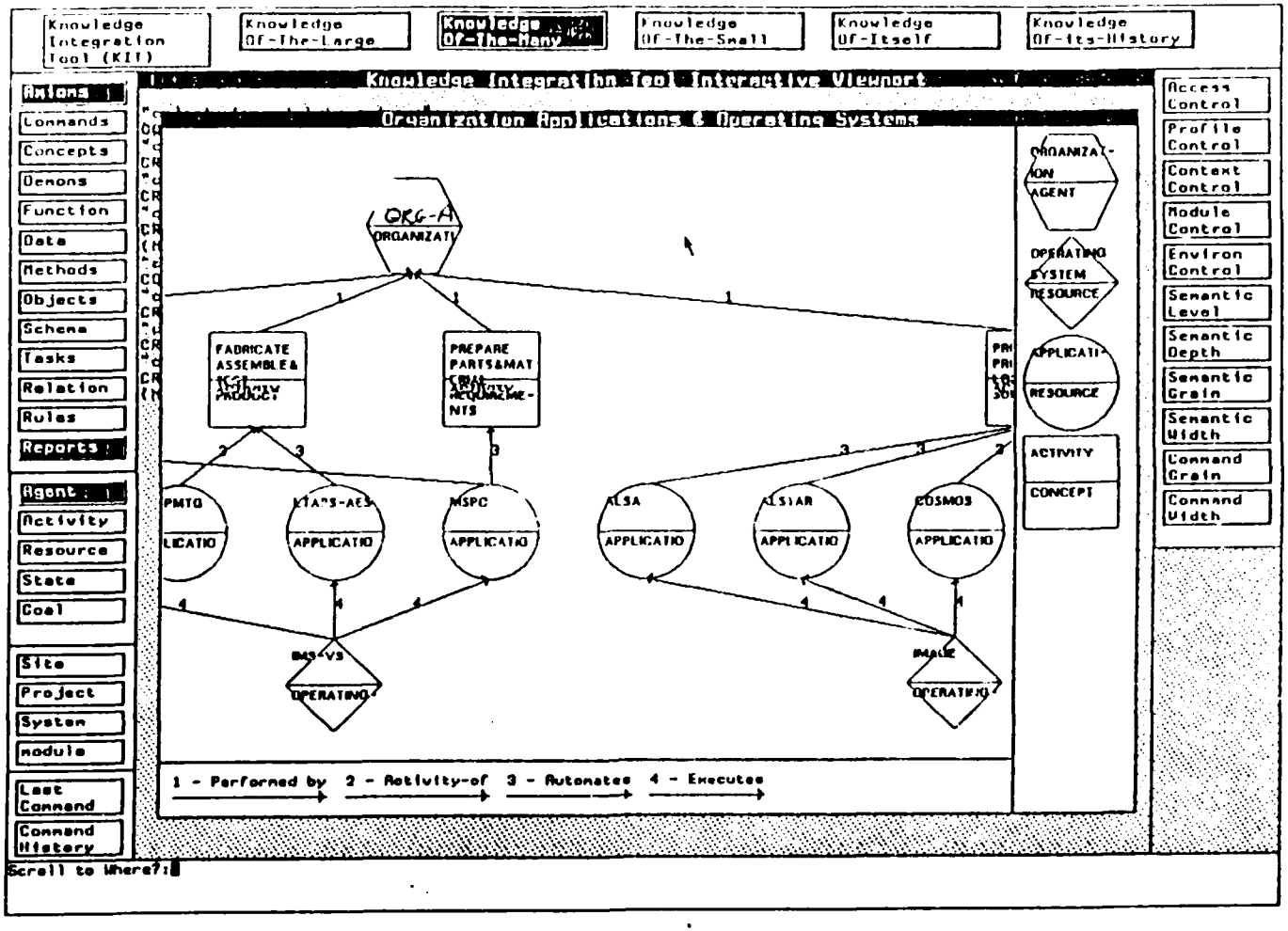
Appendix A: Organization Modelling

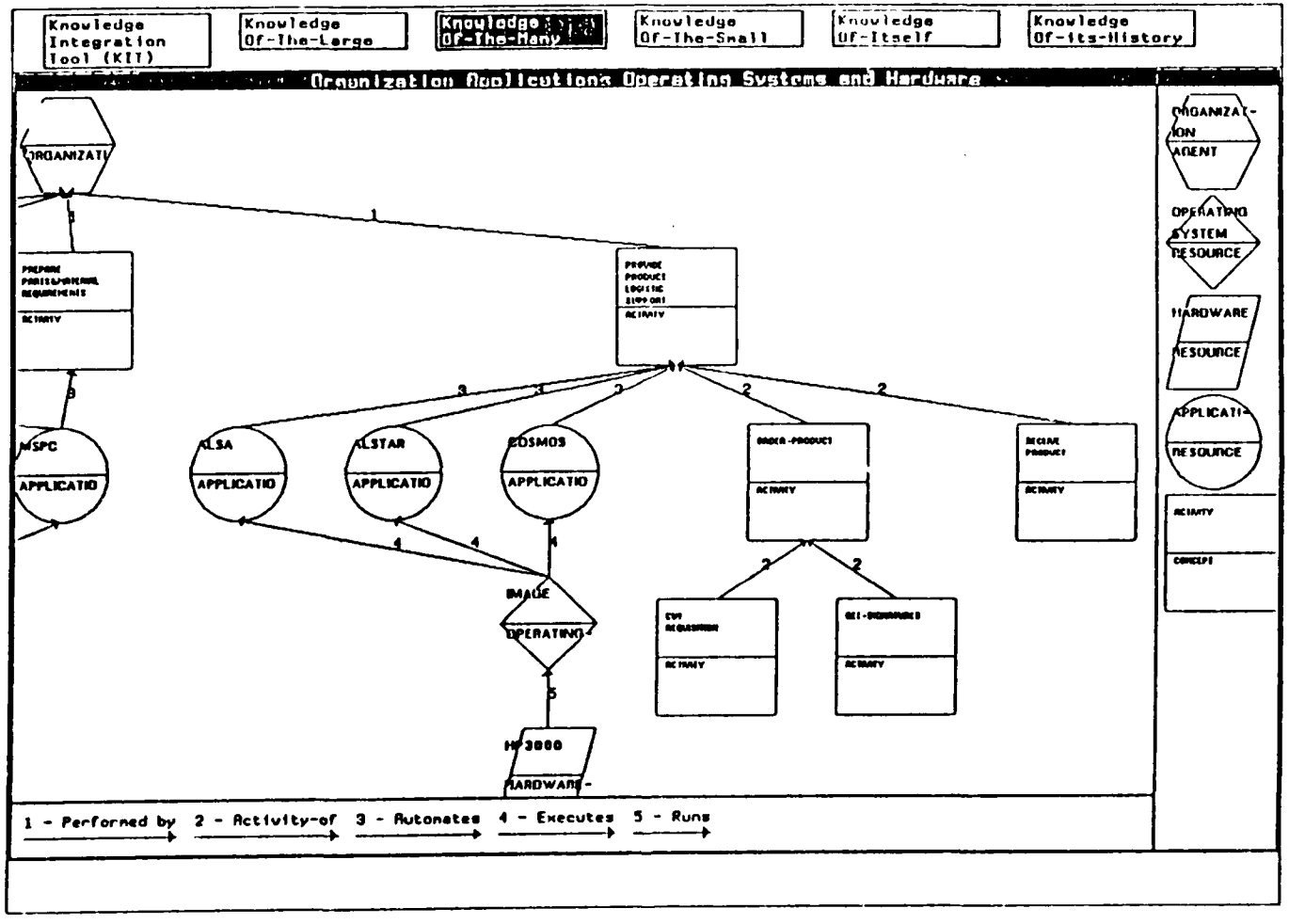
This appendix contains portions of a prototypical knowledge based application for automation planning which was created with the Knowledge Integration Tool.

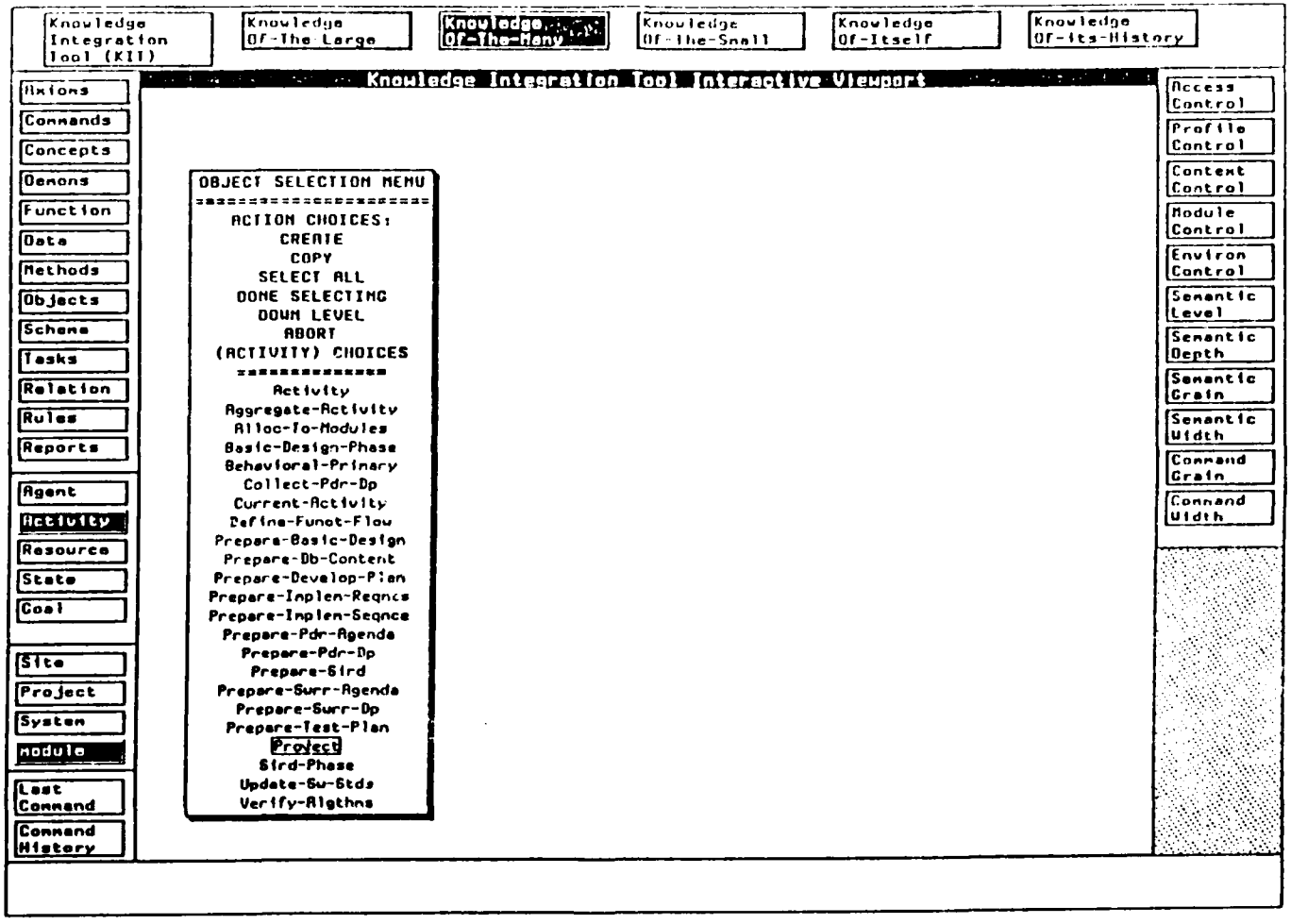

Knowledge Integration Tool (KIT)
Knowledge Of-The-Large
Knowledge Of-The-Many
Knowledge Of-The-Small
Knowledge Of-Itself
Knowledge Of-its-History
Knowledge Integration Tool Interactive Viewport
Actions
Commands
Concepts
Demons
Function
Data
Methods
Objects
Schema
Tasks
Relation
Rules
Reports
Agent
Activity
Resource
State
Goal
Site
Project
System
module
Last Command
Command History
Access Control
Profile Control
Context Control
Module Control
Environ Control
Semantic Level
Semantic Depth
Semantic Grain
Semantic Width
Command Grain
Command Width
ACTIONS FOR
METHODS OF (ORGANIZATION)
Select a command or expand command choices
Organization-Functions
Organization-Applications
Organization-Applications&Os
Organizations-Applications-Os&Hardware
Organization-Corrupted-Model
EXPAND
DONE SELECTING
ABORT

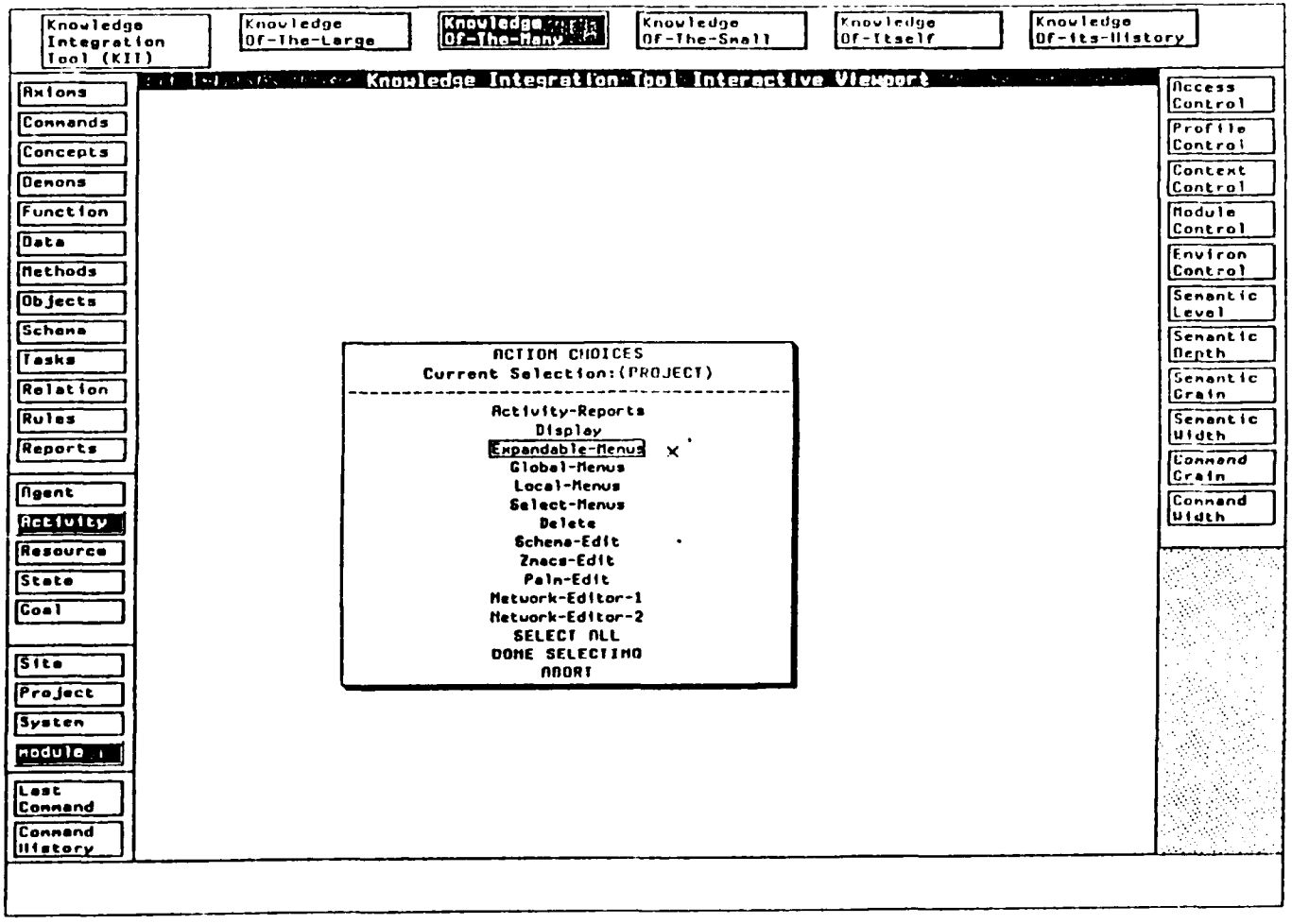

Knowledge Integration Tool (KIT)
Knowledge Of-The-Large
Knowledge Of-The-Many
Knowledge Of-The-Small
Knowledge Of-Itself
Knowledge Of-its-History
Organization Activity Hierarchy
ORG-A
ORGANIZATION AGENT
ACTIVITY
CONCEPT
PROCURE EQUIPMENT-
ACTIVITY
FABRICATE ASSEMBLE &
PRODUCT
PREPARE PARTS&MAT
PROVIDE PRODUCT
SUPPORT
ORDER PRODUCT
ACTIVITY
RECEIVE PRODUCT
ACTIVITY
GET SIGNATURES
ACTIVITY
1 - Performed by
2 - Activity-of

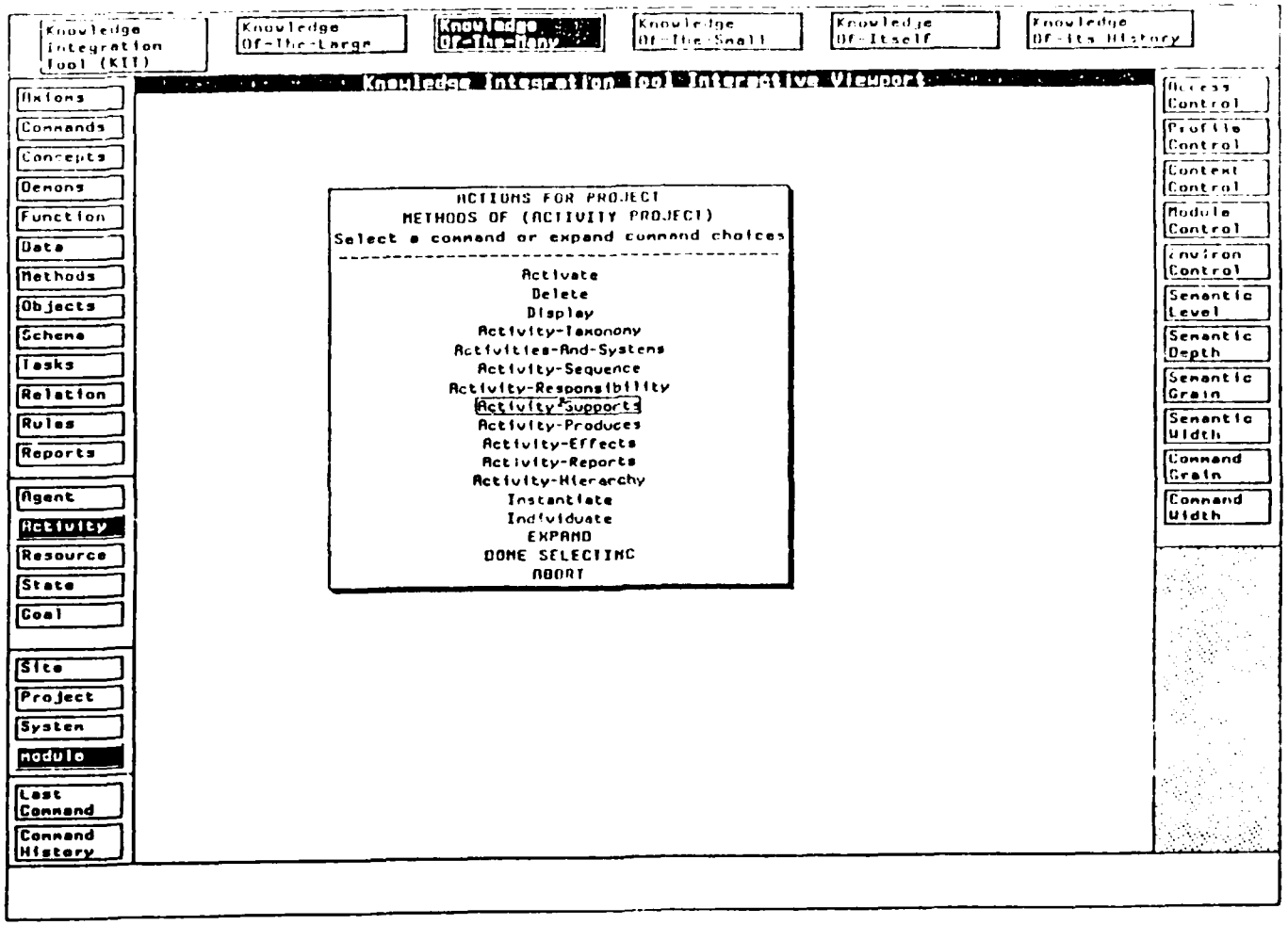

Knowledge Integration Tool (KIT)
Knowledge Of-The-Large
Knowledge Of-The-Many
Knowledge Of-The-Small
Knowledge Of-Itself
Knowledge Of-its-History
Hll Organization Applications
ORG-A
ORGANIZATI
FABRICATE ASSEMBLE&
PREPARE PARTS&MAT
ORGANIZAT-ION AGENT
APPLICATI-
RESOURCE
ACTIVITY
CONCEPT
SCPP
HELPMTG
ETAPS-AES
MSPC
ALSA
ALSTAR
COSMOS
APPLICATIO
1 - Performed by
2 - Activity-of
3 - Automates

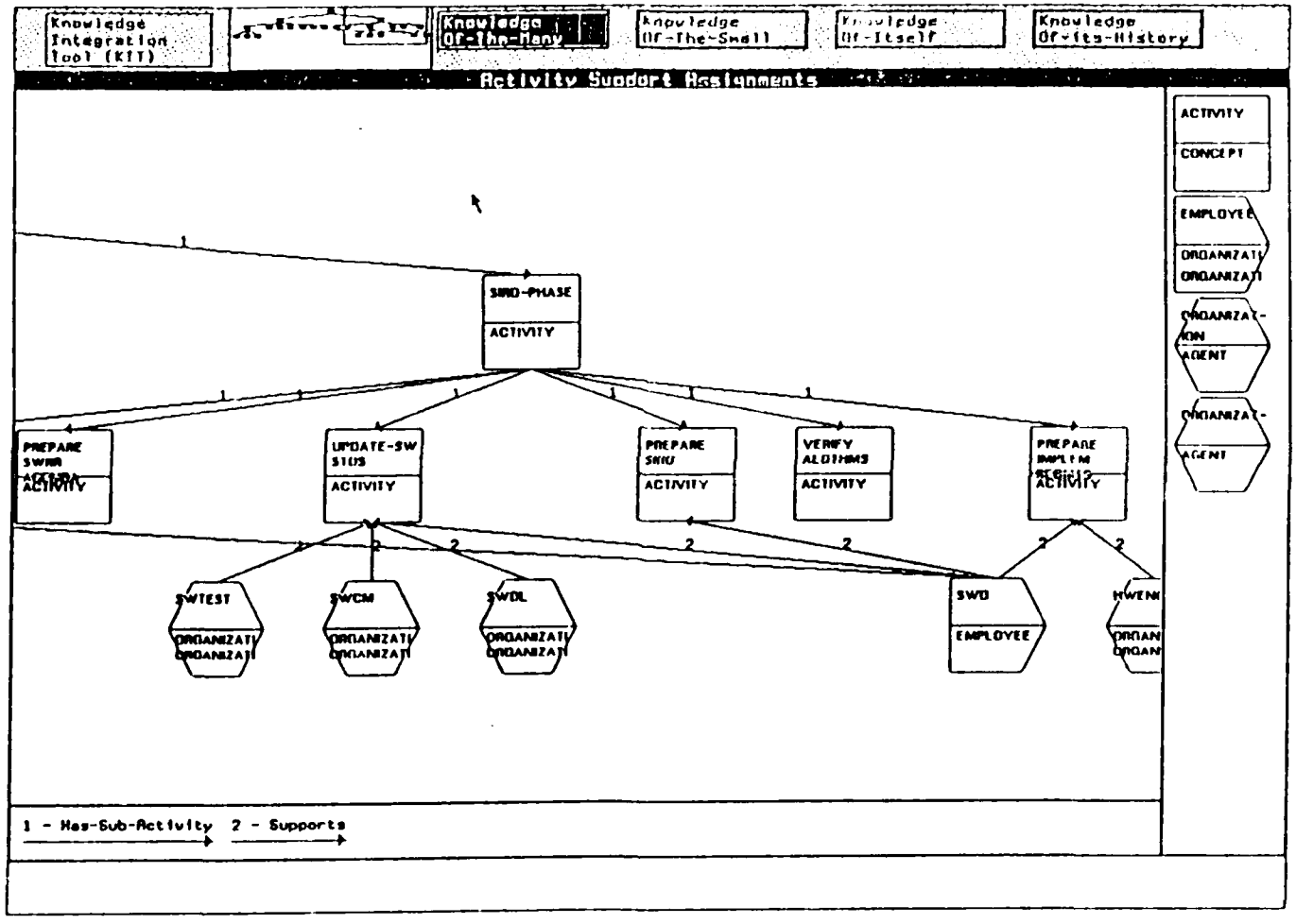

Knowledge Integration Tool (KIT)
Knowledge Of-The-Large
Knowledge Of-The-Many
Knowledge Of-The-Small
Knowledge Of-Itself
Knowledge Of-Its-History
Knowledge Integration Tool Interactive Viewport
Organization Applications & Operating Systems
Actions
Commands
Concepts
Demons
Function
Data
Methods
Objects
Schema
Tasks
Relation
Rules
Reports
Agent
Activity
Resource
State
Goal
Site
Project
System
module
Last Command
Command History
Access Control
Profile Control
Context Control
Module Control
Environ Control
Semantic Level
Semantic Depth
Semantic Grain
Semantic Width
Command Grain
Command Width
ORG-A
ORGANIZATI
FABRICATE ASSEMBLE&
PREPARE PARTS&MAT
PMTG
APPLICATIO
MSPC
APPLICATIO
ALSA
APPLICATIO
ALSIAR
APPLICATIO
COSMOS
APPLICATIO
ORGANIZATION AGENT
OPERATING SYSTEM RESOURCE
RESOURCE
ACTIVITY
CONCEPT
1 - Performed by
2 - Activity-of
3 - Automates
4 - Executes
Scroll to Where?:

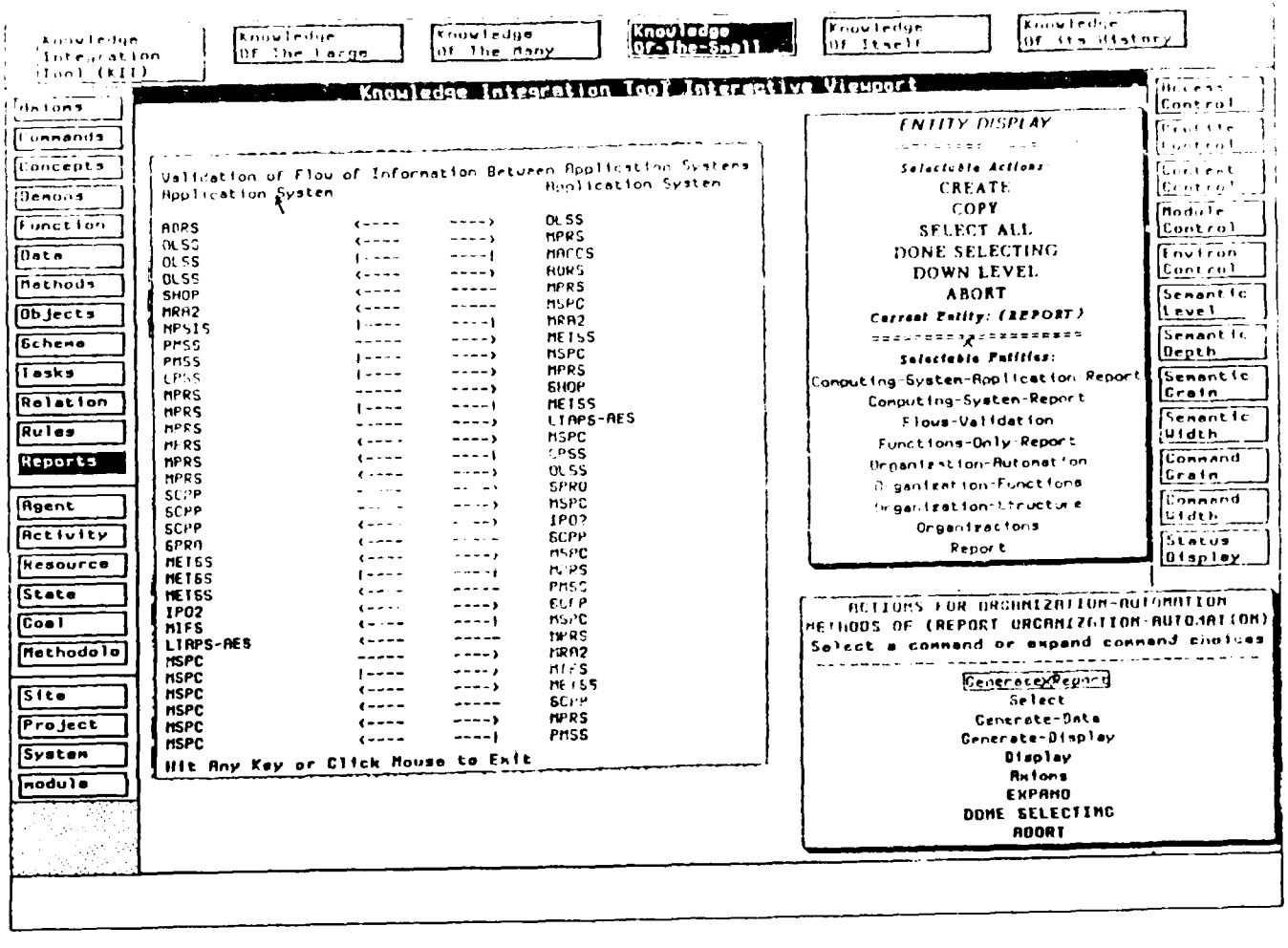

Knowledge Integration Tool (KIT)
Knowledge Of-The-Large
Knowledge Of-The-Many
Knowledge Of-The-Small
Knowledge Of-Itself
Knowledge Of-its-History
Organization Applications Operating Systems and Hardware
ORGANIZATI
PREPARE PROJECT/GENERAL REQUIREMENTS
ACTIVITY
PROVIDE PRODUCT LOGISTIC SUPPORT
ACTIVITY
MSPC
APPLICATIO
ALSA
APPLICATIO
ALSTAR
APPLICATIO
EOSMOS
APPLICATIO
ORDER-PRODUCT
ACTIVITY
SECURE PRODUCT
ACTIVITY
IMAGE
OPERATING
HP 3000
HARDWARE
ORGANIZATION AGENT
OPERATING SYSTEM RESOURCE
HARDWARE RESOURCE
APPLICATION RESOURCE
ACTIVITY
CONCEPT
1 - Performed by
2 - Activity-of
3 - Automates
4 - Executes
5 - Runs

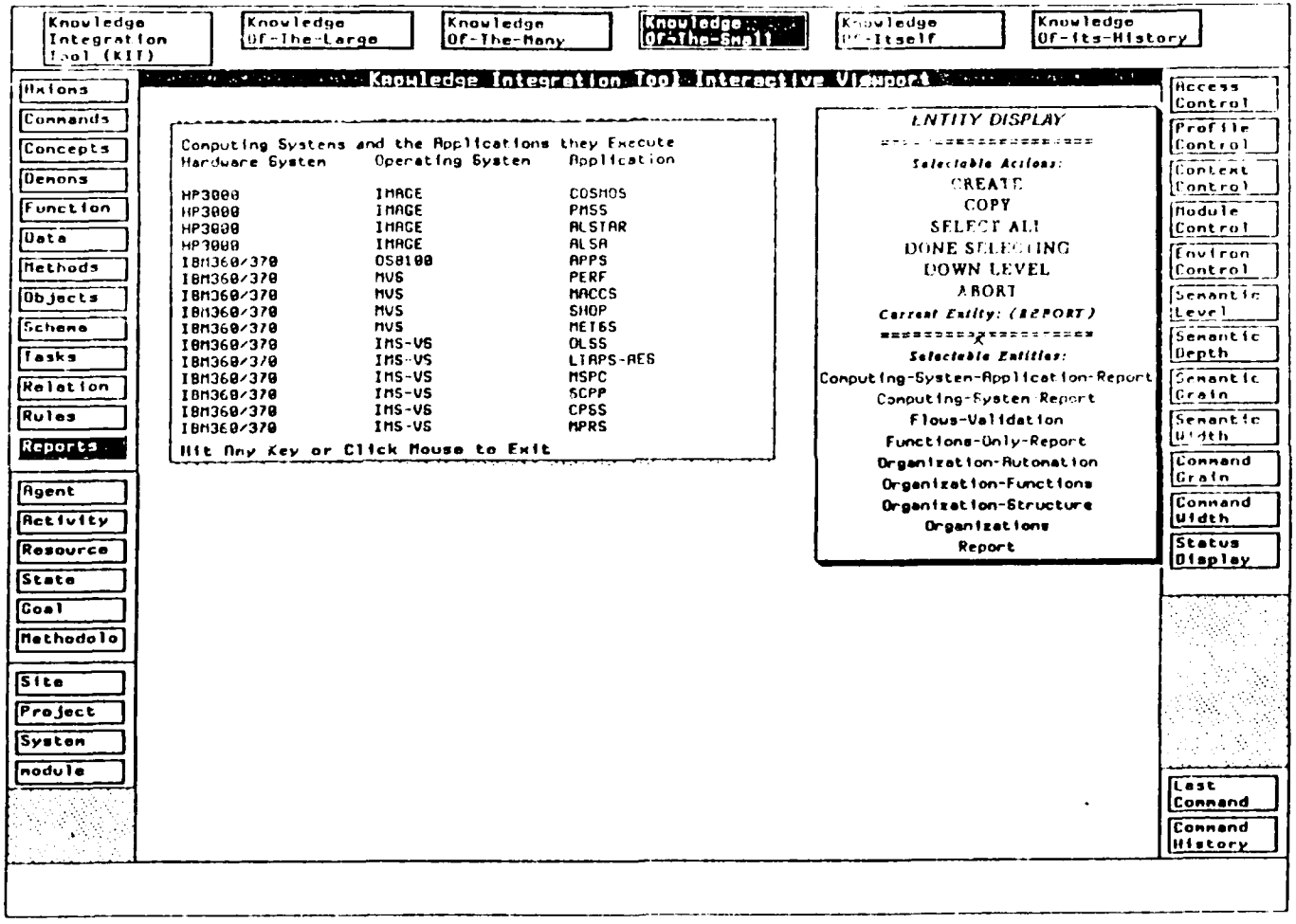

Knowledge Integration Tool (KII)
Knowledge Of-The Large
Knowledge Of-The-Many
Knowledge Of-the-Small
Knowledge Of-Itself
Knowledge Of-its-History
Knowledge Integration Tool Interactive Viewport
Axioms
Commands
Concepts
Demons
Function
Data
Methods
Objects
Schema
Tasks
Relation
Rules
Reports
Agent
Activity
Resource
State
Goal
Site
Project
System
Module
Last Command
Command History
Access Control
Profile Control
Context Control
Module Control
Environ Control
Semantic Level
Semantic Depth
Semantic Grain
Semantic Width
Command Grain
Command Width
OBJECT SELECTION MENU
ACTION CHOICES:
CREATE
COPY
SELECT ALL
DONE SELECTING
DOWN LEVEL
ABORT
(ACTIVITY) CHOICES
Activity
Aggregate-Activity
Alloc-To-Modules
Basic-Design-Phase
Behavioral-Primary
Collect-Pdr-Dp
Current-Activity
Define-Funct-Flow
Prepare-Basic-Design
Prepare-Db-Content
Prepare-Develop-Plan
Prepare-Implen-Reqmts
Prepare-Implen-Seqnce
Prepare-Pdr-Agenda
Prepare-Pdr-Dp
Prepare-Sird
Prepare-Surr-Agenda
Prepare-Surr-Dp
Prepare-Test-Plan
Project
Sird-Phase
Update-Sw-Stds
Verify-Algthms


Knowledge Integration Tool (KIT)
Knowledge Of-The-Large
Knowledge Of-The-Many
Knowledge Of-The-Small
Knowledge Of-Itself
Knowledge Of-its-History
Knowledge Integration Tool Interactive Viewport
Axioms
Commands
Concepts
Demons
Function
Data
Methods
Objects
Schema
Tasks
Relation
Rules
Reports
Agent
Activity
Resource
State
Goal
Site
Project
System
module
Last Command
Command History
Access Control
Profile Control
Context Control
Module Control
Environ Control
Semantic Level
Semantic Depth
Semantic Grain
Semantic Width
Command Grain
Command Width
ACTION CHOICES
Current Selection:(PROJECT)
Activity-Reports
Display
Expandable-Menus
Global-Menus
Local-Menus
Select-Menus
Delete
Schema-Edit
Znacs-Edit
Pain-Edit
Network-Editor-1
Network-Editor-2
SELECT ALL
DONE SELECTING
ABORT


Knowledge Integration Tool (KIT)
Knowledge Of-The-Large
Knowledge Of-The-Small
Knowledge Of-Itself
Knowledge Of-Its History
Knowledge Integration Tool Interactive Viewport
Axioms
Commands
Concepts
Demons
Function
Data
Methods
Objects
Scheme
Tasks
Relation
Rules
Reports
Agent
Activity
Resource
State
Goal
Site
Project
System
Module
Last Command
Command History
Access Control
Profile Control
Context Control
Module Control
Environ Control
Semantic Level
Semantic Depth
Semantic Grain
Semantic Width
Command Grain
Command Width
ACTIONS FOR PROJECT
METHODS OF (ACTIVITY PROJECT)
Select a command or expand command choices
Activate
Delete
Display
Activity-Taxonomy
Activities-And-Systems
Activity-Sequence
Activity-Responsibility
Activity-Supports
Activity-Produces
Activity-Effects
Activity-Reports
Activity-Hierarchy
Instantiate
Individuate
EXPAND
DONE SELECTING
ABORT


Knowledge Integration Tool (KIT)
Knowledge Of-The-Many
Knowledge Of-The-Small
Knowledge Of-Itself
Knowledge Of-Its-History
Activity Support Assignments
ACTIVITY
CONCEPT
SWD-PHASE
ACTIVITY
PREPARE SWRR
ACTIVITY
UPDATE-SW STDS
ACTIVITY
PREPARE SWDD
ACTIVITY
VERIFY ALGORITHMS
ACTIVITY
ACTIVITY
SWTEST
SWCM
SWDL
SWD
EMPLOYEE
1 - Has-Sub-Activity
2 - Supports


Knowledge Integration Tool (KIT)
Knowledge Of The Large
Knowledge Of The Many
Knowledge Of The Small
Knowledge Of Itself
Knowledge Of Its History
Knowledge Integration Tool Interactive Viewport
Actions
Commands
Concepts
Demons
Function
Data
Methods
Objects
Scheme
Tasks
Relation
Rules
Reports
Agent
Activity
Resource
State
Goal
Methodolo
Site
Project
System
Module
Access Control
Module Control
Environ Control
Semantic Level
Semantic Depth
Semantic Grain
Semantic Width
Command Grain
Command Width
Status Display
Validation of Flow of Information Between Application Systems
Application System
Application System
ADPS <---- ----> OLSS
OLSS <---- ----> MPRS
OLSS |---- ----| MACCS
OLSS <---- ----> AORS
OLSS <---- ----> MPRS
SHOP <---- ----- MSPC
MRA2 <---- ----- MRA2
MPSIS |---- ----| METSS
PMSS ----- ----> MSPC
PMSS |---- ----> MPRS
LPSS |---- ----> SHOP
MPRS ----- ----> METSS
MPRS |---- ----| LIRPS-AES
MPRS ----- ----> MSPC
MPRS <---- ----> SPSS
MPRS <---- ----| OLSS
MPRS <---- ----> SPRO
SCPP ----- ----> MSPC
SCPP <---- ----> IPO2
SCPP <---- ----- SCPP
SPRO <---- ----> MSPC
METSS |---- ----| MPPS
METSS <---- ----- PMSS
IPO2 <---- ----> SCPP
MIFS <---- ----| MSPC
LIRPS-AES <---- ----- MPRS
MSPC ----- ----> MRA2
MSPC |---- ----> MIFS
MSPC <---- ----> METSS
MSPC <---- ----- SCPP
MSPC <---- ----> MPRS
MSPC <---- ----| PMSS
Hit Any Key or Click Mouse to Exit
ENTITY DISPLAY
Selectable Actions
CREATE
COPY
SELECT ALL
DONE SELECTING
DOWN LEVEL
ABORT
Current Entity: (REPORT)
Selectable Entities:
Computing-System-Application-Report
Computing-System-Report
Flows-Validation
Functions-Only-Report
Organization-Automation
Organization-Functions
Organization-Structure
Organizations
Report
ACTIONS FOR ORGANIZATION-AUTOMATION
METHODS OF (REPORT ORGANIZATION-AUTOMATION)
Select a command or expand command choices
Generate-Report
Select
Generate-Data
Generate-Display
Display
Actions
EXPAND
DONE SELECTING
ABORT


Knowledge Integration Tool (KIT)
Knowledge Of-The-Large
Knowledge Of-The-Many
Knowledge Of-The-Small
Knowledge Of-Itself
Knowledge Of-its-History
Axions
Commands
Concepts
Demons
Function
Data
Methods
Objects
Schema
Tasks
Relation
Rules
Reports
Agent
Activity
Resource
State
Goal
Methodolo
Site
Project
System
module
Knowledge Integration Tool Interactive Viewport
Computing Systems and the Applications they Execute
Hardware System Operating System Application
HP3000 IMAGE COSMOS
HP3000 IMAGE PMSS
HP3000 IMAGE ALSTAR
HP3000 IMAGE ALSA
IBM360/370 OS8100 APPS
IBM360/370 MVS PERF
IBM360/370 MVS MACCS
IBM360/370 MVS SHOP
IBM360/370 MVS METBS
IBM360/370 IMS-VS OLSS
IBM360/370 IMS-VS LIAPS-AES
IBM360/370 IMS-VS MSPC
IBM360/370 IMS-VS SCPP
IBM360/370 IMS-VS CPSS
IBM360/370 IMS-VS MPRS
Hit Any Key or Click Mouse to Exit
ENTITY DISPLAY
Selectable Actions:
CREATE
COPY
SELECT ALL
DONE SELECTING
DOWN LEVEL
ABORT
Current Entity: (REPORT)
Selectable Entities:
Computing-System-Application-Report
Computing-System-Report
Flows-Validation
Functions-Only-Report
Organization-Automation
Organization-Functions
Organization-Structure
Organizations
Report
Access Control
Profile Control
Content Control
Module Control
Environ Control
Semantic Level
Semantic Depth
Semantic Grain
Semantic Width
Command Grain
Command Width
Status Display
Last Command
Command History

Bibliography

Boehm, B. W., Penedo, M.H., Stuckle, E. D., Pyster, A. B., "A Software Environment for Improving Productivity," IEEE , 1984.

Borgida, A., Myolopoulos, J. and Wong, H., "Generalization/Specialization as a Basis for Software Specification," in On Conceptual Modelling , New York:Springer Verlag, 1984.

Brachman R. J., "On the Epistemological Status of Semantic Networks", in Readings in Knowledge Representation , (ed. Brachman R.H and Levesque H.J.), Morgan Kaufman Publishers, Los Altos, Ca., 1985.

Brachman R. J., Fikes R. E., Levesque H. J., "Krypton: A Functional Approach to Knowledge Representation", in Readings in Knowledge Representation (ed. Brachman R.H and Levesque H.J.), Morgan Kaufman Publishers, Los Altos, Ca., 1985.

Defense System Software Development Standard , document DOD-STD-2167, and Data Item Descriptions, document, DID.

Fox M. "Organization Structuring: Designing Large Complex Software," Carnegie Mellon University, Pittsburg, PA, Dec. 1979.

----, "Organization Structuring: Designing Large Complex Software," Carnegie Mellon University, Pittsburg, PA, Dec. 1979.

----, "Constraint-Directed Search: A Case Study of Job-Shop Scheduling," (PhD Thesis), Technical Report CMU-RI-TR-83-22, Robotics Insutute, Carnegie-Mellon University, Pittsburgh, PA, 1983.

----, "The Intelligent Management System:An Overview," in Processes and Tools for Decision Support , H.G. Sol (Ed.), North-Holland, Pub. Co., 1983.

Fox, M.S., Sathi Arvind, Greenburg, M., "Issues in Knowledge Representation for Project Management," in Proceeding of IEEE Conference on Knowledge-Based Systems, Sheraton Denver Tex, Denver, Colorado, December, 1984.

Glinert, E. P, Tanimoto, S. L., "Pict: An Interactive Graphical Programming Environment," IEEE , November, 1984.

Green, C., et. al. "Report on a Knowledge-Based Software Assistant," RADC-TR-83-195, Rome Air Development Center, Griffiss AFB, NY, August 1983.

Habermann, A. N., Notkin, D.S., "The Gandalf Software Development Environment," Carnegie Mellon University, Pittsbug, PA, January, 1982.

Harandi, Mehdi T., "A Knowledge-Based Programming Support Tool," IEEE Transactions , Sept, 1983.

Hayes-Roth, Frederick. "Knowledge Based Expert Systems," IEEE Transactions , Oct., 1984.

----., "Swift: A Requirements Specification System For Software," Proceedings of IBM Tools Symposium, San Jose, CA, November, 1982.

Kedzierski, "Knowledge-Based Project Management and Communication Support in a System Development Environment," Proceeding of the 4th Jerusalem Conference on Information Technology , May, 1984.

King, Tim, "KBSA Framework," Honeywell Systems and Research Center, in KBSA Technical Exchange Meeting Proceedings , Rome Air Development Center, Grifiss AFB, NY, Sept. 1986.

Knowledge Craft , Carnegie Group Inc., Version 3.1, Vol. 1-3, August, 1986.

Matheson, James. E., "Overview of R&D Decision Analysis," in The Principles and Applications of Decision Analysis ed. Howard, R. A, and Matheson, J. E., Strategic Decisions Group., Vol 1., 1983.

Minsky, Marvin, "A Framework for Representing Knowledge", appeared in Readings in Knowledge Representation , editors Brachman, R.J., and Levesque, H.J., Morgan Kaufmann Publishers, Inc, Los Altos, CA, 1985.

Newell, A., Simon, H., "Computer Science as Empirical Inquiry: Symbols and Search", Communications of the ACM ,Vol. 19, Number 3, March, 1976.

Parnas, David L., "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering , WE-5(23), March 1979.

----., "On criteria to be used in Decomposing Systems into Modules.," CACM , 15(2), Dec. 1972.

Partsch, H. and Steinbruggen, "Program Transformation Systems," Computing Surveys , Vol. 15, No 3, September 1983.

"Project Sampler", Intelligent Systems Laboratory, The Robotics Institute, Carnegie Mellon University, Pittsburg, PA, 1984.

Quillian, J. L., "Semantic Memory", PhD Th., Carnegie Mellon University, Pittsburg, PA, 1966.

Ramamoorthy, C. V., Prakash, A., Tsai, W., Usuda, Y., "Software Engineering: Problems and Perspectives," IEEE Transactions on Software Engineering, October, 1984.

Sathi, A, Fox, M., Greenburg, M., Morton, T.,"Callisto An Intelligent Project Management System", Carnegie Mellon University, Pittsburg, PA, 1984.

Sathi, A., Fox, M.S. and M. Greenbug, "Representation of Activity Knowledge in Project Management", IEEE Transactions on PAMI , Sept, 1985.

Schank, R. C., Rieger, C. J., "Inference and the Computer Understanding of Natural Language", Artificial Intelligence 5(4) , 1974.

Smith, R.G, Dinitz, R., Barth, P., "Impulse-86: A Substrate for Object-Oriented Interface Design," ACM , 1986.

Stucki, L. G., Walker, H.D., "Concepts and Prototypes of ARGUS," Proceedings of the Symposium on Software Engineering Environments , Lahnstein, Germany, June, 1980, in: Software Engineering Environments , edited by Horst Huenke, North-Holland Publishing Company, Amsterdam - New York -Oxford, 1981.

Tichy and Ward, "A Knowledge-Based Graphical Editor," submitted to ACM Transaction on Graphics , special issue on user interfaces, Carnegie Group, Inc., July, 1986.

Tichy, Walter F., "Software Development Control Based on System State Descriptions," Ph.D., dissertation, Carnegie-Mellon University, Computer Science Department, January, 1980.

Waters, R. C., "The Programmer's Apprentice: A Session with KBEmacs," IEEE Transactions on Software Engineering , Vol. SE-11, No. 11, November 1985.

Wirth, N., "Program Development by Stepwise Refinement," CACM Vol. 14, No. 4.

Woods, W. A., "What's in a link? Foundations for semantic networks", in Representation and Understanding , D.G. Bobrow and A.M. Collins (eds.), Academic Press: New York, , 1984.

PROJECT MANAGEMENT ASSISTANT: WHAT HAVE WE LEARNED?

Lt Kevin M. Benner, USAF and Louis J. Hoebel
Command and Control Software Technology Division
Rome Air Development Center
Griffiss AFB, NY 13441-5700

INTRODUCTION

In 1983 Rome Air Development Center (RADC) published a report on a Knowledge-Based Software Assistant (KBSA) [2]. This document brought together key ideas on how artificial intelligence (AI) could be used in the software development process. Since then RADC has embarked on the first of three contract iterations to develop both a Knowledge-Based Software Assistant and the enabling supporting technologies. This paper focuses on one facet of the KBSA, the Project Management Assistant (PMA). The goals of the PMA are to provide knowledge-based help to the development team in project communication, coordination, and task management. The purpose of this paper is to describe what we have learned from PMA, which concluded in December 1986. Before describing what has been learned from the PMA effort a foundation must first be laid. This foundation consists of a description of PMA and the supporting technologies. Following this will be an analysis of what we have gained from the PMA effort both in terms of life cycle activities and supporting technologies.

SUPPORTING TECHNOLOGIES

The supporting technologies fall into four main categories: a wide spectrum language, general inferential systems, domain specific inferential systems, and system integration.

A wide spectrum language (WSL) is a single language which provides the user with the ability to capture the formal semantics of the system under development regardless of the level of detail (or the step in the development cycle). A wide spectrum language is both a language and an environment. It must provide uniform expressibility, regardless of what is being described (ie. requirements, specifications, code, test cases, project management policy, etc). Not only must a WSL be able to express these objects, it must do so in a way which is consistent at all levels, both syntactically and semantically.

A general inferential system is a system which supports reasoning. In particular, we are concerned with the overall efficiency of this reasoning, how to capture such things as logic in inference rules and data structures, the quality of explanation generated by the system, and the ability to apply this inferencing power to specific domains.

Domain specific inferential systems extend general inferential systems to include aspects unique to software development. This topic focuses on the knowledge representation of software development objects and inference rules and, in particular, how these objects and rules can be formally represented and used for further reasoning. More specifically, this category can be broken down further into three subcategories: formal semantic model of the software development environment, knowledge representation and management of software development objects, and specialized inferential systems which incorporate rules that have been tuned to a specific problem or task.

System integration deals with the inherent competition between facets and how a technology base can be put together such that all phases in the software development process are supported sufficiently.

PMA OVERVIEW

Work on the PMA formalism definition and construction of a prototype began in 1985 [3, 4, 5]. Kestrel Institute was the developer. The life cycle goals of PMA were to provide knowledge-based help to users and managers in project communication, coordination, and task management.

The capabilities of PMA fall into three categories: project definition, project monitoring, and user interface. Project definition consists of structuring the project into individual tasks and then scheduling and assigning these tasks. Once the project has been decomposed into manageable tasks, it must be monitored. This monitoring is in the form of cost and schedule constraints. Also included in monitoring is the enforcement of specific management policies (eg. DoD-Std-2167, rapid prototyping, KBSA, etc.). In addition, PMA provides a good user interface for project monitoring and project definition. This interaction is in the form of direct queries/updates, Pert charts, and Gantt charts.

The above capabilities were important, but would be expected of any project management tool. What sets PMA apart from its predecessors is the expressibility and flexibility of the PMA architecture. Not only does PMA handle user defined tasks, but it also understands their products and the implicit relationships between them (eg. components, tasks, requirements, specification, source code, test cases, test results, and milestones). Also present in PMA are objects unique to programming-in-the-large (ie. versions, configurations, derivations, releases, and people).

From a technical perspective, the advances made in PMA include: the formalization of the software development objects enumerated above, the development of a powerful time calculus for representing temporal relationships between software development objects [6], and a mechanism for directly expressing and enforcing project policy.

PMA AND SUPPORTING TECHNOLOGIES: WHAT HAVE WE LEARNED?

The results of the PMA effort have been twofold. First, PMA has pulled at the supporting technologies such that they have advanced the state of these technologies. Secondly, PMA has made progress in formalizing how

project management can be accomplished.

With respect to advances in formalizing project management activities, PMA has formalized the products of software development (eg. tasks, components, milestones, requirements, specifications, test cases etc.) from a project management perspective and provided a language to describe the process by which these products came about. In the current software engineering literature this is described as "process programming"[7]. This concept has been a part of PMA since its inception, though not referenced by any specific name. PMA has demonstrated that a software project can be described formally and reasoned about.

The following sections will describe the advances in the support technologies that PMA has made.

A Wide Spectrum Language: The WSL used for the PMA prototype is REFINE. REFINE is best described as a programming language that provides constructs for specification in a variety of styles (eg. functional, logical, procedural, and object oriented). These constructs include: user defined object classes, set-theoretic data types, constructs from first order logic, relations defined by assertions, transforms, a pattern language, and conventional programming constructs [8].

As well as providing the above constructs, REFINE gives the developer the ability to define new constructs. These new constructs are types of assertions used to maintain knowledge base integrity. One example is CHECKING, which is used as a trigger to monitor for particular condition(s) and then flags the system when they arise. COMPLAINING, often used in conjunction with CHECKING, interacts with the project manager to explain what has gone wrong. MAINTAINING is another type of assertion which automatically insures a given condition remains true.

Because of the expressibility and the extensibility of PMA's WSL, REFINE, PMA has demonstrated that a WSL can handle in a uniform manner both software programming constructs and process programming constructs within a single language.

Domain Specific Inferential Systems: PMA's impact on domain specific inferential systems is seen in its characterization of software development objects in the PMA knowledge base. This consists of how to represent requirements, specification, code, development history, project policies, etc. such that they may be reasoned about from a project management perspective. This reasoning consists of enforcing policy (more accurately an activities coordination role), project structuring, task assignment, task scheduling, schedule monitoring, task monitoring, and cost monitoring.

General Inferential Systems: Within the area of general inferential systems, much work has been done outside of KBSA and PMA took advantage of that work. One area that has benefited from PMA research is that of time representation and manipulation.

The calculus is an extension of James Allen's interval calculus for reasoning about time [1]. Allen's relational algebra has 13 atoms and is therefore of finite size. The expressive power, in terms of time representation, of PMA's time calculus (developed by Peter Ladkin) comes

from its infinite algebra [6]. The temporal logic primitives are implemented directly in REFINE.

Some of the advantages cited by Peter Ladkin of the Kestrel team include the capturing of the metaphysics of time and presenting a natural way for representing time. Also, the representation allows for the joining of seconds into minutes and days into weeks or months such that there is a linear hierarchy of standard units.

System Integration: Finally, within the area of system integration, PMA, being the first facet on contract and without the advantage of a common framework, relies on a tightly coupled relationship between facets. Knowledge base objects, which would have come from other facets, are assumed to be present and up to date in the PMA knowledge base.

SUMMARY

Although the PMA is a prototype intended as a technology demonstration, it will be used at both RADC and at the Software Productivity Consortium for experimental purposes. The next developmental phase will be to build a functional implementation of the current research and to integrate it into a conventional software environment. The purpose of this is two fold: 1) to demonstrate transfer of KBSA technology into the operational world as that technology comes to maturation and 2) to continue work in AI and project management in real world applications. This work will begin in late 1987.

This initial prototype concentrated on providing a knowledge-based framework. It did not concentrate on interaction with the underlying KBSA support system such as policy enforcement for version and access controls of objects developed by other facets. Nor was emphasis placed on a common user interface or PMA integration into a standard work environment. These latter two issues are important for user acceptance and the former issues are related to the overall KBSA development, cost and performance. These are some of the issues which will be explored and answered in the next PMA development and its integration into a conventional software development environment.

REFERENCES

[1] Allen, J.F., "Maintaining Knowledge about Temporal Intervals", Comm. A.C.M.26(11), November 1983, 832-843.

[2] Green, C. et al., " Report on a Knowledge-Based Software Assistant," RADC Tech. Report TR-83-195.

[3] Gilham, L., "KBSA-PMA Program Specification", Documentation, RADC Contract F30602-84-C-0109.

[4] Gilham, L., "KBSA-PMA User Manual", Documentation, RADC Contract F30602-84-C-0109.

[5] Jullig, R., et al., "KBSA-PMA Technical Report", Final Technical Report, RADC, Nov. 1986.

[6] Ladkin, P., "Primitives and Units for Time Specification", Proceedings of AAAI-86, Philadelphia, PA, 11 -15 August 1987.

[7] Osterweil, L., "Software Processes Are Software Too", Proceedings of 9th Annual ICSE, Monterey, CA, 30 March - 2 April 1987.

[8] Reasoning Systems, "REFINE User's Guide", June 15, 1986.

Simplifying the Construction of Domain-Specific Automatic Programming Systems: The NASA Automated Software Development Workstation Project*

Bradley P. Allen
Peter L. Holtzman

Inference Corporation
5300 W. Century Blvd.
Los Angeles, CA 90045

Abstract

We provide an overview of the Automated Software Development Workstation Project, an effort to explore knowledge-based approaches to increasing software productivity. The project focuses on applying the concept of domain-specific automatic programming systems (D-SAPS) to application domains at NASA's Johnson Space Center. We describe a version of a D-SAPS developed in Phase I of the project for the domain of Space Station momentum management, and discuss how problems encountered during its implementation have led us to concentrate our efforts on simplifying the process of building and extending such systems. We propose to do this by attacking three observed bottlenecks in the D-SAPS development process through the increased automation of the acquisition of programming knowledge and the use of an object-oriented development methodology at all stages of program design. We discuss how these ideas are being implemented in the Bauhaus, a prototype CASE workstation for D-SAPS development.

1. Increasing software productivity through domain-specific automatic programming

Software development is an increasingly serious bottleneck in the construction of complex automated systems. Increasing the reuse of software designs and components has been viewed as an important way to address this problem, possibly increasing productivity by an order of magnitude or more [9]. A promising approach to achieving software reusability is through *domain-specific automatic programming.*

---

* This is a revised version of a paper to appear in the Proceedings of the Space Operations Automation and Robotics Workshop, Houston TX, August, 1987. This work is supported by NASA Lyndon B. Johnson Space Center under Contract NAS 9-17766

Domain-specific automatic programming systems (D-SAPS) use domain knowledge to automate the refinement of a program description (written in a high-level domain language) into compilable code (written in a procedural target language) [1]. D-SAPS can be distinguished from the more traditional domain-independent automatic programming systems in that the specification of the program is in a domain-specific language accessible to an end user, rather than a formal specification language (e.g. the predicate calculus with equality). Application generators of the type used in business report generation (e.g. Focus and DBASE-II) are examples of D-SAPS in which the refinement process is completely automatic and implemented procedurally [19]. More complex domains can be handled if the user is allowed to interact with and control the refinement process. Prototype knowledge-based systems that support user interaction and which work for practical application domains have been successfully developed (e.g. Draco [16], ΦNIX [3], and KBEmacs [25]).

2. The Automated Software Development Workstation Project

Since the fall of 1985, Inference Corporation has been involved in an effort, sponsored by NASA's Lyndon B. Johnson Space Center, to explore the applicability of domain-specific automatic programming to NASA software development efforts. Phase I of the project focused on the development of a D-SAPS for the domain of Space Station momentum management [19]. During Phase I, A prototype D-SAPS was constructed, comprised of:

- a components catalog of FORTRAN subroutines used in the construction of Space Station orbital simulations;
- a design catalog of programs implemented using the system;
- a interactive graphical design system using a dataflow specification language for design editing and components composition;
- code generators for component interfaces, numerical subroutines and main programs; and
- a rule-based expert that:
  - proposed refinements for unimplemented modules in the dataflow

specification:

- flagged inconsistencies at manually-specified component interfaces; and
- suggested possible workarounds to "patch" inconsistencies (e.g. coordinate system conversion routines).

The system was implemented by hand, to serve as a model for the implementation of similar D-SAPS for other NASA domains. The functionality and performance of the prototype was adequate to demonstrate the applicability of the D-SAPS approach to software development at NASA JSC. However, our experience in building this system led us to agree with other D-SAPS developers in noting that:

- "domain analysis and design is *very hard*" [16]; and
- "domain-specific systems can be quite useful within their range of application, but the range is often quite narrow" [2].

We feel that these two issues must be addressed if D-SAPS are to play a significant role in future software development environments. Therefore, in Phase II of the project, we are attempting to address the bottlenecks in the D-SAPS development process that lead to these observations.

3. Addressing the bottlenecks in D-SAPS development

We have focused on three bottlenecks in the D-SAPS development process that were observed during the development of the Phase I system:

- developing a domain language;
- describing design refinements and constraints; and
- describing the generation of target language code from a sufficiently detailed program description.

We plan to reduce the effort spent on each of these tasks by:

- [illegible] the programming knowledge acquisition process; and
- [illegible] an object-oriented development methodology at all stages of the program design process.

We are currently implementing a knowledge-based D-SAPS development workstation,

and the Bauhaus[*] system which exploits these two approaches. We now describe how to design CAD Bauhaus systems using these approaches.

3.1. Automating the programming knowledge acquisition process

By structuring the design process so that the types of knowledge required are made explicit, the knowledge acquisition process can be made simpler [4], and the resulting knowledge base easier to modify [9]. To this end we are using a problem-solving architecture based on that of the RIME [24] and SOAR [12] systems, implemented using the ART expert system building tool [11]. This architecture allows us to organize design knowledge into a hierarchy of *problem spaces*, representing program design tasks. Each problem space consists of a set of operators for performing the task represented by the space. In the Bauhaus, a problem space is associated with each program description that the system has in its knowledge base. The task represented by the problem space is that of refining the program description until it is sufficiently detailed to allow a code generator to translate it into code in the target language. The design process in the Bauhaus occurs in the following way:

1. **Select the initial design problem**: the user copies and edits an initial program description from the set of program descriptions in the knowledge base using a structure editor, making it the *current description*. The initial problem space is that associated with refining this description.

2. **Propose operators**: the Bauhaus determines what operators are applicable to the current description.

3. **Choose an operator**: the Bauhaus chooses an operator from the proposed set using operator preferences and constraints associated with the problem space, and implemented as ART production rules. User interaction is requested when the system reaches an *impasse*: when no operator applies, when the system is unable to derive a preference for a specific operator, or when the system's preferences are inconsistent [12]. This interaction takes one of two forms:

    - the user chooses a proposed operator for the system to pursue;

---

[*] Our system is named after the German institute of art founded in 1919 by the architect Walter Gropius. The Bauhaus was noted for a program that synthesized technology, craftsmanship and artistic aesthetics.

- the user edits the current program description, in which case we return to step 2.

4. **Apply the chosen operator**: the Bauhaus applies the chosen operator to the current description. The operator may:

   - traverse into a problem subspace,
   - refine the current description,
   - signal that the task for the problem space is complete, or
   - signal that the task cannot be successful completed.

   We then return to step 2.

The design process terminates when the top-level task of refining the initial program description is successfully completed. This occurs when the description is detailed enough to allow the generation of target language code to occur. Given this problem-solving architecture, we now discuss the knowledge acquisition mechanisms used to obtain the descriptions, operators, operator preferences and constraints used in the design process.

3.1.1. Acquiring descriptions

Our representation of domain objects and operations uses a description language, implemented in ART schemata, that is similar to KRYPTON [17]. New descriptions of domain objects and operations are created from existing descriptions using the copy-and-edit technique espoused by Lenat in the CYC system, [13] and are placed in the appropriate location in a subsumption hierarchy by an automatic classifier [8]. This use of description copy-and-edit together with automatic classification reduces the effort required to extend the domain language used to describe systems, by fostering reuse of existing domain languages in the creation of new domain languages. Using a subsumption hierarchy of descriptions as the organizing framework for the [illegible] of objects and operations supports user access for copy-and-edit actions [illegible] a browse-and-reformulation browser similar in design to ARGON [18]. Reformulation will permit a naive user of the Bauhaus to find a description [illegible] copy-and-edit action more easily than using a tradition query mechanism

3.1.2. Acquiring operators, operator preferences, and constraints

[illegible] preferences are acquired [illegible] constraints are acquired when [illegible] the application of an operator, causing the Bauhaus to backtrack [illegible]. This type of knowledge acquisition [illegible] by the user can be characterized as [illegible] approach [illegible] [15]. In this respect, the Bauhaus is similar to the VEXED VLSI design system [22].

3.2. Using an object-oriented development methodology

By using object-oriented design (OOD) [5], we can decrease the level of effort required to implement a code generator that takes a sufficiently detailed program description and produces compilable target language code. This is due to the natural correspondence between the world and its model in an object-oriented framework [7]. In the Bauhaus, the world is the set of target language software components and [illegible] templates and the model is the set of descriptions of objects and sequences of operations in an application program. Ada's language level support of abstract data types and the existence of commercially supported reusable software component libraries constructed using OOD principles [6] make it our first choice as a target language in the Bauhaus system. The mapping between the description of an program and its realization in Ada [illegible] and the generation of the main subprogram in which the program objects are [illegible]. [illegible] the Bauhaus could easily be extended [illegible] OOD [illegible] ART [illegible]

4. System status and limitations

Implementation of the Bauhaus is currently underway using ART running on a Symbolics Lisp machine under the Genera 7.1 environment. Support for Ada compilation and library management is provided by the Symbolics Ada programming environment. As of July 1987, ART-based representations for descriptions, operators, operator preferences and constraints have been designed, the problem-solving architecture and basic knowledge acquisition algorithms have been designed and implemented, and the target language reusable components library has been selected. The user interface is currently under construction, and the domain analysis for the demonstration domain, orbital flight simulation, is underway. We plan to demonstrate the use of the Bauhaus in the construction of a D-SAPS for this domain in the first quarter of 1988.

In the current design of the Bauhaus, there are a number of issues relevant to D-SAPS that we do not address:

- **Lifecycle issues**: the Bauhaus is only useful as a programming-in-the-small environment, and ignores programming-in-the-large issues (e.g. version control). These would have to be addressed in a production-quality system.

- **Persistent object bases**: the Bauhaus has no provision for saving session state in a more sophisticated manner than simply saving changes out to a text file. We are looking to object-oriented database management systems to provide an answer here [4].

- **Automated algorithm synthesis**: the Bauhaus will always reach an impasse if a programming task requires algorithm design. However, the architecture should be extensible to encompass this kind of problem solving (e.g., see the work by Steier on the Cypress-Soar and Designer-Soar algorithm design systems [21]).

5. Conclusion

[illegible] evidence that domain-specific automatic programming is a viable approach to [illegible] software productivity. To make this approach a practical one, the task of [illegible] constructing D-SAPS must be made simpler. As described above, we plan to [illegible] in improving the knowledge acquisition and software engineering

methodologies used in constructing D-SAPS. Our ultimate goal is a production-quality system that could be described as an "application generator generator"; i.e., a knowledge-based environment for the construction of special-purpose systems for the generation of applications software by end-users. Such a system could be available to systems analysts and designers in a DP/MIS organization for use when an applications programming task occurs frequently enough to merit the creation of a D-SAPS.

References

1. Barstow, D. "Domain-Specific Automatic Programming". *IEEE Transactions on Software Engineering 11*, 11 (November 1985).

2. Barstow, D. Artificial Intelligence and Software Engineering. Proceedings of the 9th International Conference on Software Engineering, IEEE, March-April, 1987.

3. Barstow, D., Duffey, R., Smoliar, S., and Vestal, S. An overview of ΦNIX. Proceedings of the Second National Conference on Artificial Intelligence, AAAI, August, 1982.

4. Bernstein, P.A. Database System Support for Software Engineering. Proceedings of the 9th International Conference on Software Engineering, IEEE, March-April, 1987.

5. Booch, G. "Object-Oriented Development". *IEEE Transactions on Software Engineering 12*, 2 (February 1986).

6. Booch, G. *Software Components With Ada*. Benjamin/Cummings Publishing, 1987.

7. Bordiga, A., Greenspan, S., and Mylopoulos, J. "Knowledge Representation as the Basis for Requirements Specification". *Computer 18*, 4 (April 1985).

8. Brachman, R.J. and Levesque, H.J. The Tractability of Subsumption in Frame-Based Description Languages. Proceedings of the National Conference on Artificial Intelligence, AAAI, August, 1984.

9. Horowitz, E. and Munson, J.B. "An Expansive View of Reusable Software". *IEEE Transactions on Software Engineering 10*, 5 (September 1984).

10. Horowitz, E., Kemper, A., and Narasimhan, B. Application Generators: Ideas for Programming Language Extensions. Proceedings of ACM'84 Annual Conference: The Fifth Generation Challenge, ACM, October, 1984.

11. Inference Corporation. *ART 3.0 Reference Manual*. Inference Corporation 1987.

12. Laird, J.E., Newell, A. and Rosenbloom, P.S. "SOAR: An Architecture for General Intelligence". *Artificial Intelligence 33*, 1 (1987).

13. Lenat, D.B., Prakash, M., and Shepherd, M. "CYC: Using Common Sense Knowledge To Overcome Brittleness and Knowledge Acquisition Bottlenecks". *AI Magazine 6*, 4 (Winter 1986).

14. Marcus, S., McDermott, J., and Wang, T. Knowledge Acquisition for Constructive Systems. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, August, 1985.

15. Mitchell, T.M., Mahadevan, S., and Steinberg, L.I. LEAP: A Learning Apprentice for VLSI Design. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, August, 1985.

16. Neighbors, J.M. "The Draco Approach to Constructing Software from Reusable Components". *IEEE Transactions on Software Engineering 10*, 5 (September 1984).

17. Patel-Schneider, P.F. Small can be Beautiful in Knowledge Representation. Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems, December, 1984.

18. Patel-Schneider, P.F., Brachman, R.J., and Levesque, H.J. ARGON: Knowledge Representation meets Information Retrieval. Proceedings of the First Conference on Applications of Artificial Intelligence, IEEE, December, 1984.

19. Prouty, D.A. and Klahr, P. Automated Software Development Workstation. Proceedings of the Conference on AI for Space Applications, NASA, November, 1986.

20. Soloway, E., Bachant, J. and Jensen, K. Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a VERY Large Rule Base. Proceedings of the National Conference on Artificial Intelligence, AAAI, July, 1987.

21. Steier, D.M., Laird, J.E., Newell, A., Rosenbloom, P.S., Flynn, R.A., Golding, A., Polk, T.A., Shivers, O.G., Unruh, A. and Yost, G.R. Varieties of Learning in Soar: 1987. Proceedings of the Fourth International Workshop on Machine Learning, June, 1987.

22. Steinberg, L.I. Design as Refinement Plus Constraint Propagation: The VEXED Experience. Proceedings of the National Conference on Artificial Intelligence, AAAI, July, 1987.

23. Tou, F.N., Williams, M.D., Fikes, R., Henderson, A., and Malone, T. RABBIT: An Intelligent Database Assistant. Proceedings of the Second National Conference on Artificial Intelligence, AAAI, August, 1982.

24. Van de Brug, A., Bachant, J. and McDermott, J. "The Taming of R1". *IEEE Expert 1*, 3 (Fall 1986).

25. Waters, R.C. "The Programmer's Apprentice: A Session with KBEmacs". *IEEE Transactions on Software Engineering 11*, 11 (November 1985).

ASDL - AN INTELLIGENT ACQUISITION SUPPORT TOOL

Richard Adler and Terry Gleason
The MITRE Corporation, Bedford, MA 01730

ABSTRACT

This paper describes ASDL, an experimental prototype for an AI-based acquisition support environment. Acquisition analysts and system engineers will employ ASDL to represent and maintain functional requirements and other aspects of system specifications in the form of symbolic models. These symbolic system descriptions are constructed by copying and adapting (historical) models of previous systems or elements from a library of generic function, component, and system templates. Analysis capabilities are under development to assist users in evaluating the economic and technical feasibility of system models. Specifically, facilities are being implemented to represent and dynamically simulate behavioral aspects of systems and to compute system sizing estimates.

INTRODUCTION

Government acquisition programs follow a standardized development cycle, from concept exploration and formal system requirements definition through design, implementation and field operation. As systems become larger and more complex, maintenance and analysis of the information needed to characterize systems through the acquisition cycle becomes increasingly difficult and costly.

A variety of computerized automation tools already exist to support system development activities. Unfortunately, these tools are not easily integrated because they operate on different kinds of models. Documentation tools manipulate simple text and graphics. Simulators and design tools model system structures and relations (e.g., connectivity, data and control flow). Costing and sizing tools manipulate algebraic attributes for system components. These models are often implicit (i.e., neither visible nor explicitly manipulable), and are rarely extensible to represent information that lies outside of a given tool's immediate scope.

MITRE's Automated System Development Library is a prototype for an AI-based tool designed to assist Government personnel in early phases of acquisitions (i.e., through preliminary design). The objective of ASDL is to assist acquisition analysts and system engineers to represent, maintain, and analyze symbolic models of system specifications. ASDL models depict system architecture,

functionality and behaviors within a single, explicit representational framework. Extensions to encompass system design factors (e.g., estimated size and complexity, performance needs and projections, maintainability, reliability, and similar engineering constraints), are also being developed.

The ASDL prototype's maintenance capabilities include creation, modification, browsing, and storage of system description models. The analysis utilities under investigation will help acquisition analysts to assess the technical and economic feasibility of systems as depicted in these models.

This paper provides an overview of ASDL. First, the tool's library-based approach to system model construction and current representational capabilities are described. Behavioral modeling and the architecture of the simulator shell are discussed next. ASDL's current analysis capabilities and user interface are then reviewed, followed by a sketch of planned enhancements.

BACKGROUND

This section reviews the kinds of information called for by DoD-STD-2167 for specification products in early phases of system acquisitions. The objective of ASDL is to capture these aspects of system descriptions in explicit structures in symbolic models.

The System/Segment Specifications (A-Specs/SSS), produced by Government customers (and/or agents such as MITRE), defines a system's mission, operational modes and functional requirements. Requirements for interfacing to existing systems, quantitative performance needs, and other design constraints are also delineated. Design constraints include quality factors such as reliability and maintainability, security, logistics, and operating environment(s).

In source selection, the Government evaluates proposals submitted in response to the functional requirements specification. Proposals outline technical approach, size and cost estimates, engineering and management methodologies.

Winning contractors submit a sequence of specifications and design descriptions for Government approval before implementing a system. Software Requirements Specifications (B-Specs/SRS) allocate functions among particular hardware and software subsystems in a high-level system architecture and specify data flow relations among those subsystems. The Software Top Level Design Document (STLDD) presents Preliminary Design interfaces and protocols between software and hardware subsystems. Software Detailed Design Documents (SDDD) specify data structures and data and control flows within subsystems.

LIBRARY-BASED SYSTEMS MODELING

New systems are seldom totally unique; typically, they incorporate existing capabilities, whole systems and system fragments, enhancing or adapting borrowed items as necessary. ASDL reflects this fact by incorporating two libraries containing reusable models and model elements to facilitate the generation of new system descriptions.

The first library is being filled with historical system descriptions - ASDL models for previous acquisition programs and for completed (early) phases of ongoing system development efforts. Current contents include model fragments for system specifications created or reviewed by MITRE.

The second library is being populated with generic or prototypical system functions and components. ASDL's current library focuses on Communications, Command, Control, and Intelligence (C^3I) systems, modeling: standard C^3I functions (e.g., data base services, signal and message processing); system components (e.g., sensors, communications interfaces, computers); systems (e.g., communications networks); data objects (e.g., messages, signal pulses, computer system account profiles); and systems personnel (e.g., operators, maintenance staff, users). Alternative generic libraries can be inserted to support template-based modeling of different system engineering domains.

Figure 1 displays a portion of the ASDL generic library, organized as a frame hierarchy. Each tree node designates an individual frame template. The lines connecting nodes in the figure represent class-subclass relations between library frames.

ASDL models are expressed in a frames language (KEE). Frames represent systems, components, functions, and the relations that hold between them. Frame slots depict attributes descriptive of object classes, such as component size and complexity factors and physical or performance characteristics. Figure 2 displays a generic library template for radar systems.

Slot facets describe attribute value cardinality, units and legal values restrictions. The legal values descriptor for the Baud.Rate attribute of the Communications.Device component class, for example, is (ONE.OF 300 1200 2400 ...). ONE.OF is a term in KEE's Boolean "ValueClass" language interpreted as "at least one of." The value of the units facet for Baud.Rate is "bits per second." Additional ASDL facets hold text strings for supplemental explanations of attributes or attribute values and for references to relevant Government standards documents. Explicit representation of such information in a standardized format facilitates the development of automated analysis capabilities.

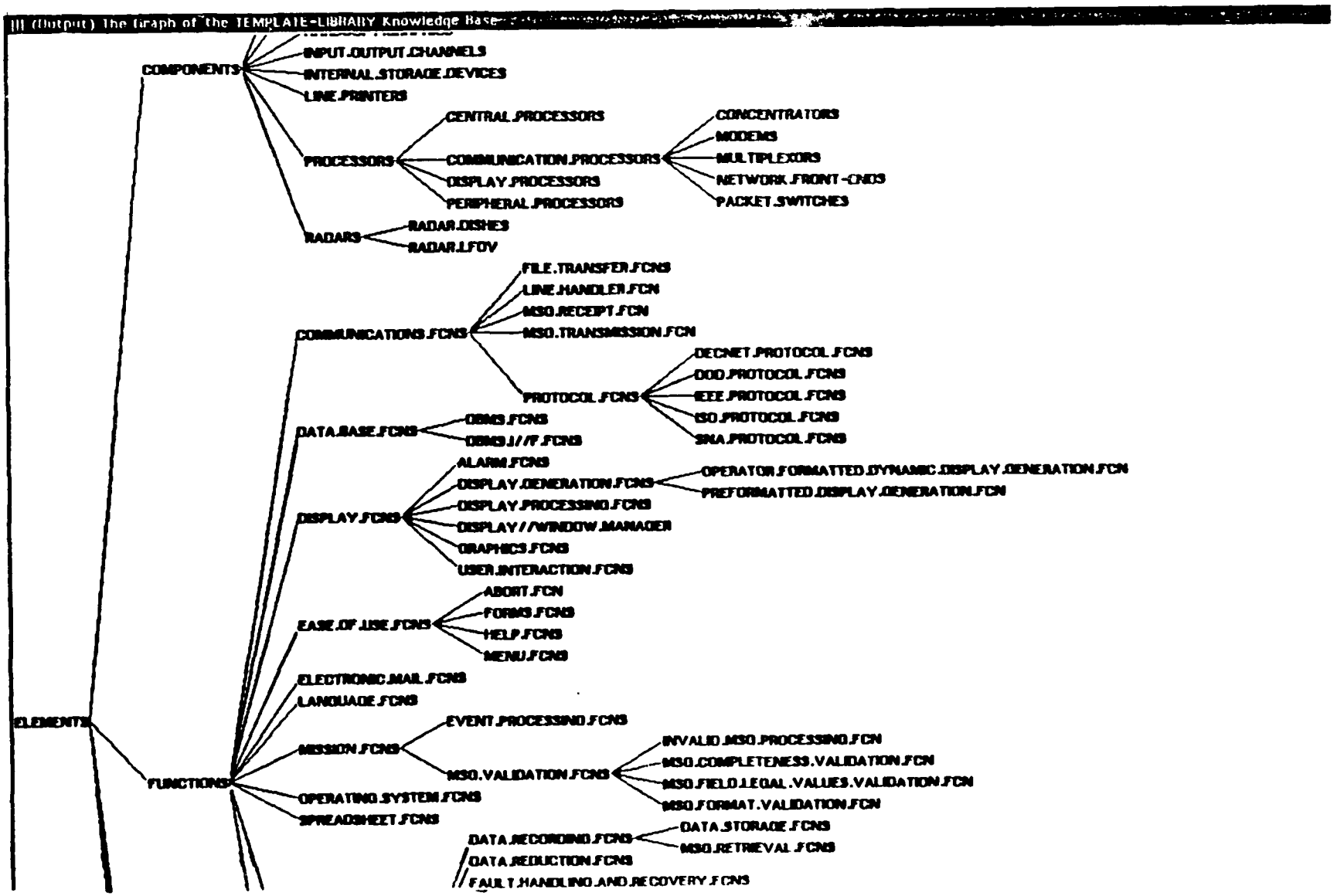
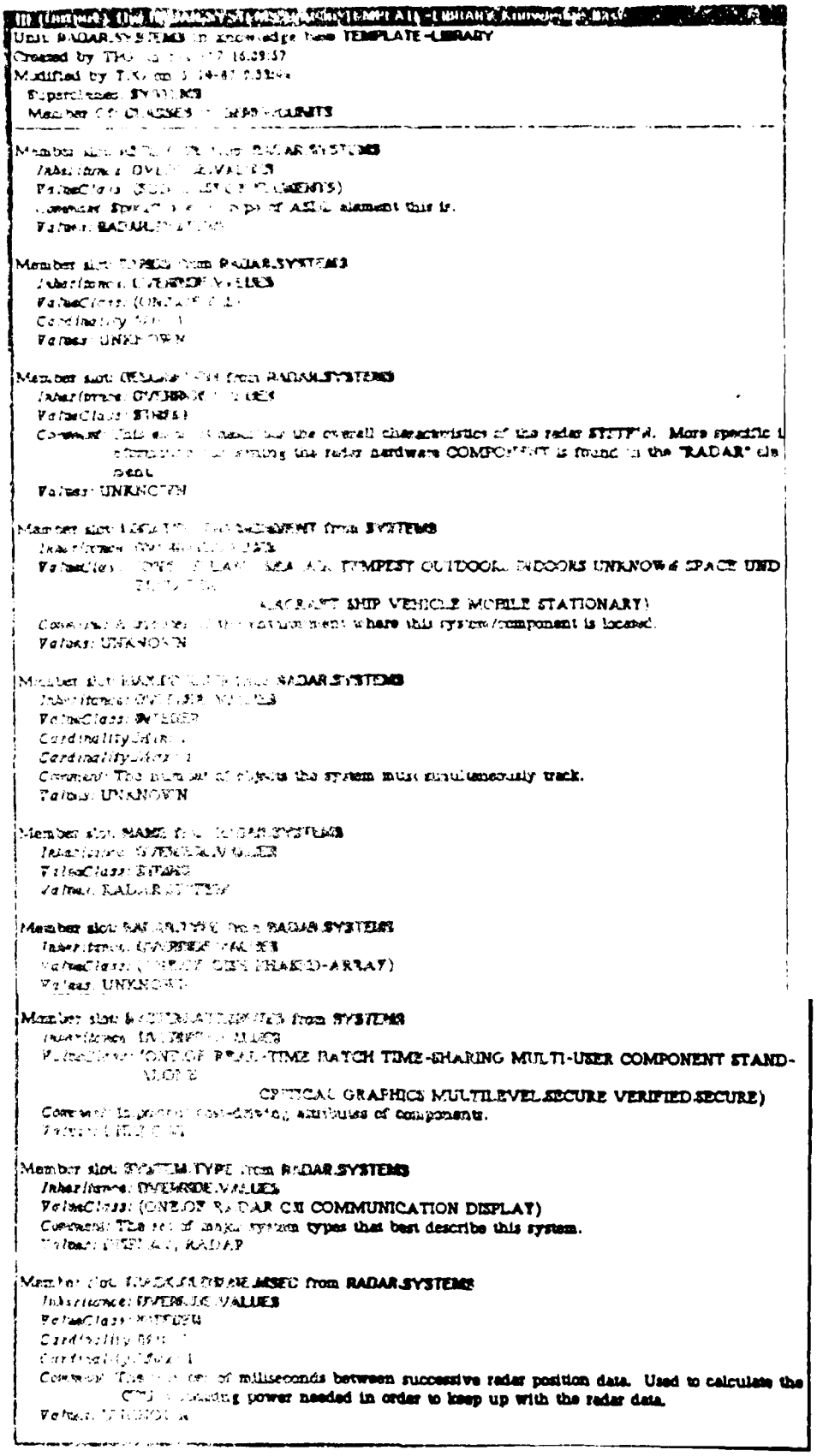


Figure 1. Portion of ASDL Generic Modeling Library



Radar.Systems Generic Modeling Library Template

Developmental system models, or knowledge bases, are generated through a simple copy-and-edit strategy. Users select appropriate frames from ASDL's libraries and copy them into the new model workspace. Copied templates and system fragments are then customized to reflect the exact content of the system description under development. Customization proceeds by supplying values to existing slot attributes or by defining new attributes and then specifying values.

Figure 3 displays a frame for an SSS for a radar system customized from the generic template shown in Figure 1. Note that the line identifying the slot name (e.g., "Member Slot: BANDS from RADAR-XYZ"), indicates the source object (viz., RADAR-XYZ), for the slot value. Sometimes values are inherited from library templates (e.g., RADAR-SYSTEMS). Note also the unfilled slots - many attributes do not receive values until well into the sequence of contractor specifications (e.g., descriptions of network protocol suites).

After library frames have been customized, relational connections between function and component frames are specified. ASDL's predefined relations include: data and control flow; hardware and software interfaces (internal and external); structural and functional decompositions; allocation of functions to components; and mappings of functions across system models. Figures 4b and 5b show instances of functional and structural decompositions of functional areas and subsystems depicted in figures 4a and 5a, respectively. Users can define additional relations by specializing (i.e., creating subclasses of), existing relation templates in the generic modeling library.

System architecture is represented via structural connectivity relations. Specifically, interface relation instances connect instances of customized system components (or systems). Component instances are linked rather than classes because acquisitions typically require multiple copies of system components (e.g., workstations), and of whole systems (e.g., at multiple sites). Figures 4a and 5a display the radar system's functional and structural interfaces. The darkened boxes indicate components external to the subsystem being defined.

Many relational connections between system elements are too complex to be represented by simple "pointer" relations. Consider, for example, the requirement that system component security levels (and functionality) have to match the security classifications of data objects that the system handles. Another example is the relational requirement that system processor speeds match bandwidths of their associated data buses.

```
Unit RADAR-XYZ in knowledge base RADAR-XYZ
Created by TPG on 5-27-87 13:07:55
Modified by TPG on 5-27-87 12:09:54
  Superclasses: RADAR.SYSTEMS
  Member: Of: CLASSES in GENERICUNITS

  A real-time, 2-band (S & X), sea-based,phased-array antenna, radar system built in xxxx. The sizing infor
mation was obtained by examining program listings, statistics, and summaries.

Member slot: ASDL.TYPE from RADAR.SYSTEMS
   Inheritance: OVERRIDE.VALUES
   ValueClass: (SUBCLASS.OF ELEMENTS)
   Comment: Specifies what type of ASDL element this is.
   Values: RADAR.SYSTEMS

Member slot: BANDS from RADAR-XYZ
   Inheritance: OVERRIDE.VALUES
   ValueClass: (ONE.OF S X)
   Cardinality.Min: 1
   Values: S, X

Member slot: DESCRIPTION from RADAR.SYSTEMS
   Inheritance: OVERRIDE.VALUES
   ValueClass: STRING
   Comment: This element describes the overall characteristics of the radar SYSTEM. More specific informatio
         n concerning the radar hardware COMPONENT is found in the "RADAR" element.
   Values: UNKNOWN

Member slot: LOCATION.ENVIRONMENT from RADAR-XYZ
   Inheritance: OVERRIDE.VALUES
   ValueClass: (ONE.OF LAND SEA AIR TEMPEST OUTDOORS INDOORS UNKNOWN SPACE UNDERWATE
         R AIRCRAFT SHIP
                  VEHICLE MOBILE STATIONARY)
   Comment: Attributes of the environment where this system/component is located.
   Values: SEA, SHIP, MOBILE

Member slot: MAX.OBJECTS from RADAR.SYSTEMS
   Inheritance: OVERRIDE.VALUES
   ValueClass: INTEGER
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: The number of objects the system must simultaneously track.
   Values: UNKNOWN

Member slot: NAME from RADAR-XYZ
   Inheritance: OVERRIDE.VALUES
   ValueClass: STRING
   Values: RADAR-XYZ

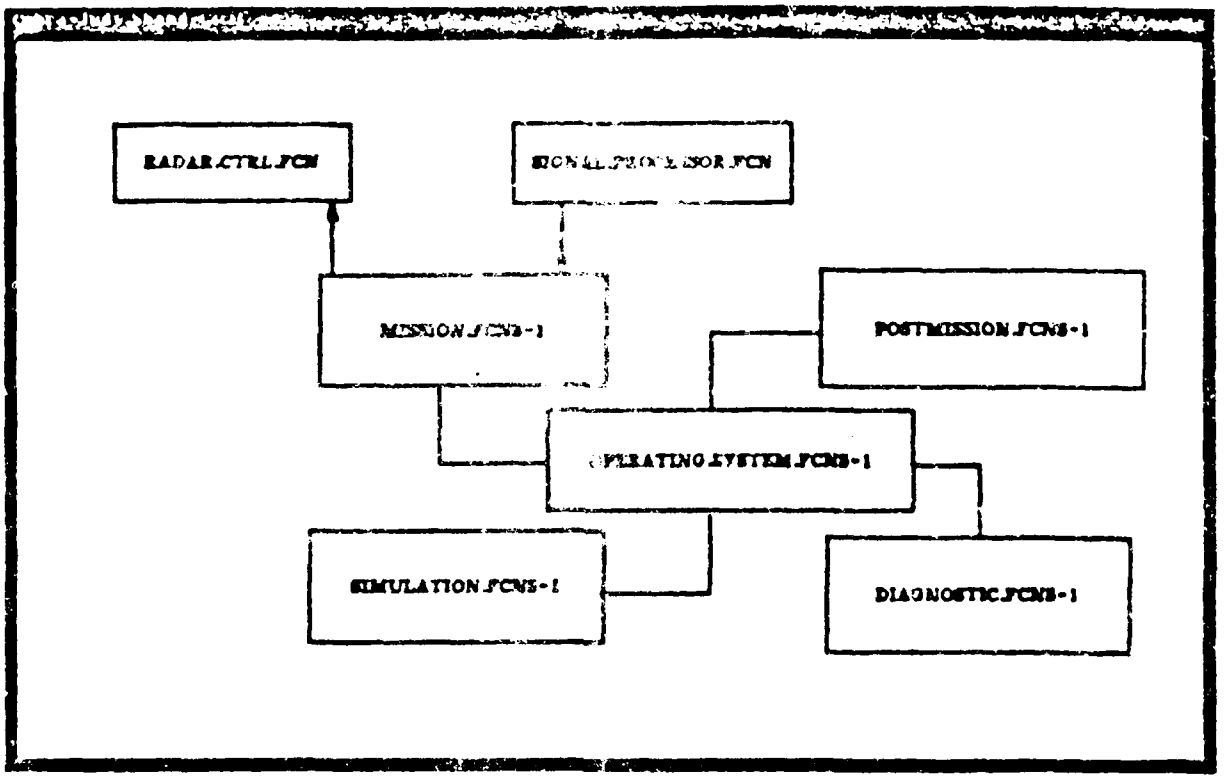
Member slot: RADAR.TYPE from RADAR-XYZ
   Inheritance: OVERRIDE.VALUES
   ValueClass: (ONE.OF DISH PHASED-ARRAY)
   Values: PHASED-ARRAY

Member slot: SYSTEM.ATTRIBUTES from RADAR-XYZ
   Inheritance: OVERRIDE.VALUES
   ValueClass: (ONE.OF REAL-TIME BATCH TIME-SHARING MULTI-USER COMPONENT STAND-ALONE
         CRITICAL GRAPHICS
                  MULTILEVEL.SECURE VERIFIED.SECURE)
   Comment: Important cost-driving attributes of components.
   Values: REAL-TIME, STAND-ALONE, GRAPHICS

Member slot: SYSTEM.TYPE from RADAR.SYSTEMS
   Inheritance: OVERRIDE.VALUES
   ValueClass: (ONE.OF RADAR C3I COMMUNICATION DISPLAY)
   Comment: The set of major system types that best describe this system.
   Values: DISPLAY, RADAR

Member slot: TRACK.CLOSURE.MSEC from RADAR.SYSTEMS
   Inheritance: OVERRIDE.VALUES
   ValueClass: INTEGER
   Cardinality.Min: 1
   Cardinality.Max: 1
   Comment: The number of milliseconds between successive radar position data. Used to calculate the CPU pr
         ocessing power needed in order to keep up with the radar data.
   Values: UNKNOWN
```

Figure 3. Radar Application System (Customized Library Template)



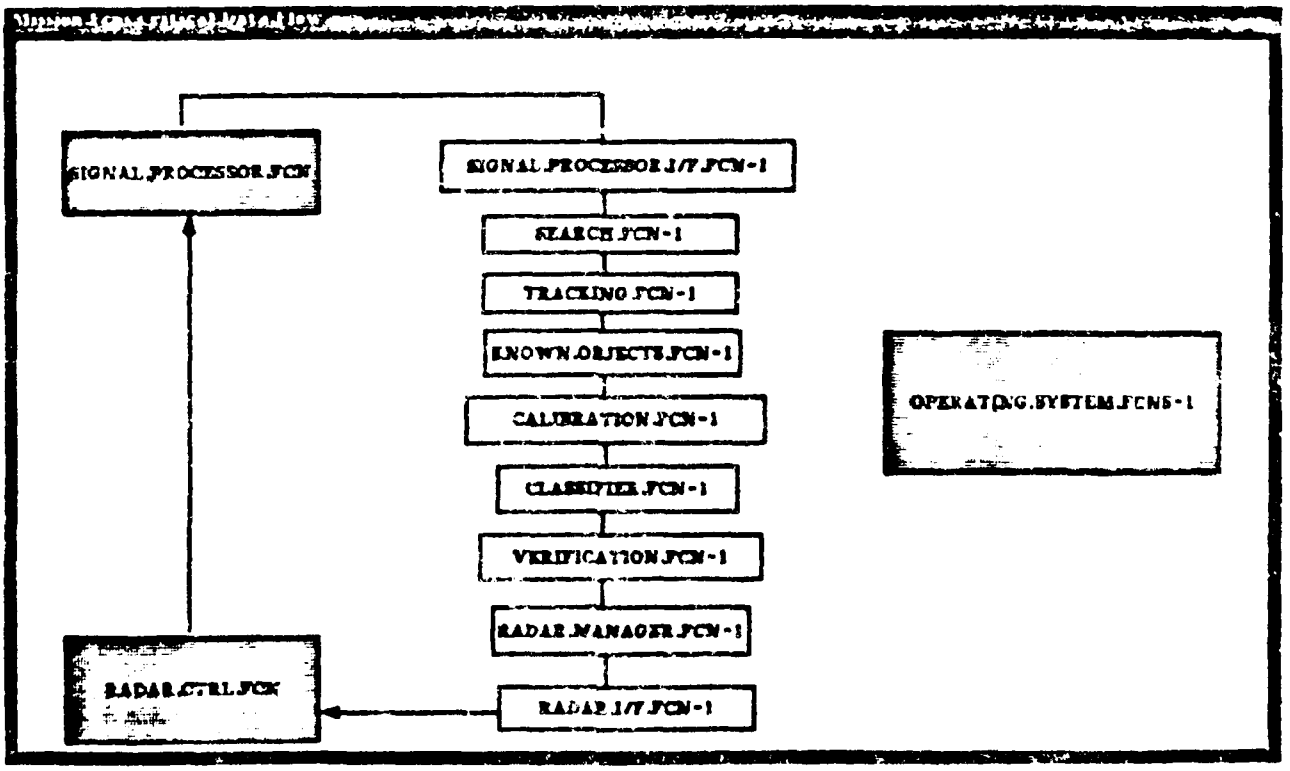


Figure 4a. Radar Application System Functional Areas

Figure 4b. Radar Application System Functional Decomposition of Mission Processing Functional Area

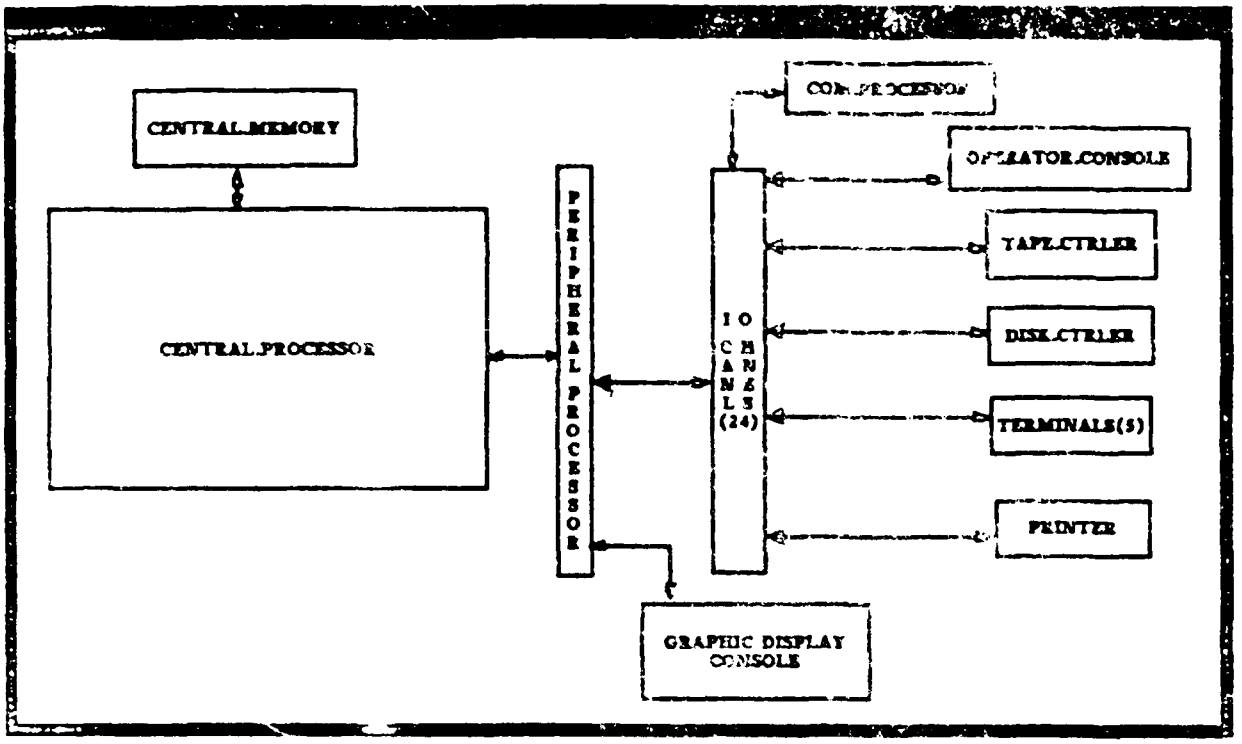


Figure 5a. Radar Application System Architecture

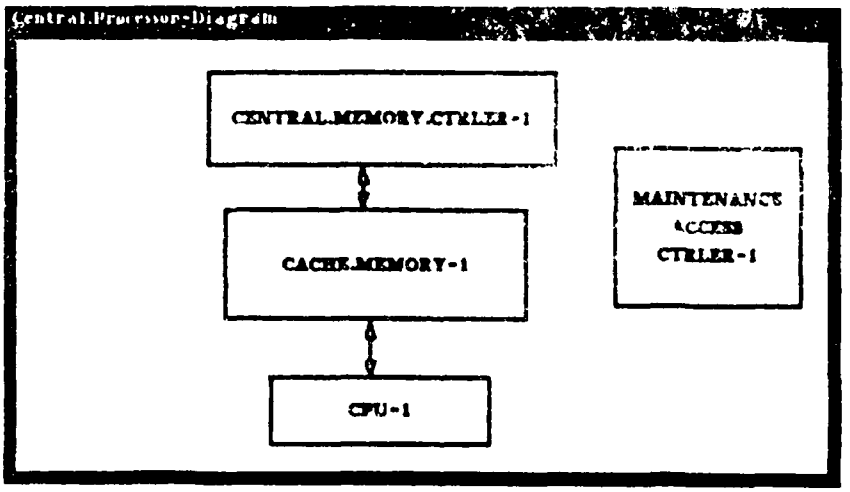


Figure 5b. Structural Decomposition of Radar Application System Central.Processor

ASDL employs a [illegible] constraint language for expressing such relations. An example [illegible] constraint is:

IF the RELIABILITY [illegible] and
the FUNCTION.CRITICALITY [illegible] HIGH
THEN the MODE of [illegible] EMBEDDED

Constraints in ASDL [illegible] particular objects (e.g., CPU#12), or classes of objects [illegible] FUNCTIONS. If objects referenced in the constraint satisfy the [illegible] antecedent clauses, but not the consequents, the constraint is violated, and users are notified of the problem.

BEHAVIORAL MODELING

Behavior, here, refers to the collections of event sequences taking place within systems, which realize specific functional capabilities. That is, functions act to induce one or more changes of state into a system, its subsystems and their structures, and the data objects that a system processes. ASDL includes mechanisms for representing and simulating these dynamic system function behaviors.

Briefly, ASDL's simulator shell consists of frames representing a synchronous clock, a simulation manager, a scheduler, and a test scenario generator. The simulation mechanisms are driven by object-oriented procedural attachments, which are implemented via slots in KEE's hybrid frames language.

Users construct behavioral system models by supplying the following ingredients: a description of flows between system functions; a description of system prioritization and scheduling algorithms; and descriptions of function actions on data objects and/or the model system and its components.

Functional flows specify sequencing, branching and looping relations between system functions. Branching and looping conditions specify combinations of system and data object attribute values (e.g., operational modes, target priorities), that determine flow (e.g., whether a track is stored or both stored and displayed).

ASDL will support modeling of both concurrent and sequential processes. System (or component server) processing priorities consist of queue sorting conditions. Examples include First-In-First-Out and rank ordering based on data object type (e.g., command vs. track), or attribute values. Concurrency modeling in ASDL depends both on functional flows and on resource allocation data - server numbers and their nature (e.g., single vs. multichannel) and function behavior requirements (i.e., what servers are specified).

Function behaviors depict the simulated outcomes of executing functions such as operational mode switch commands or data processing. For early acquisition models, behaviors typically consist of changes to values of attributes for system objects (e.g., components, data). More complex actions can also be depicted (e.g., sorting, collating, correlating). ASDL is being equipped with a menu-based interface for declarative specification of behaviors; but explicit programming will be required to represent more sophisticated function actions, such as specific algorithms or display screen mock-ups.

A behavioral model is exercised by loading test scenarios (sequences of exogenous events) into the simulator shell. Examples of test events include signal trains or messages being injected through external system interfaces and simulated user-initiated command sequences. Given these stimuli, the behavioral simulation generates (endogenous) events, representing applications of system functions to data objects and model system responses to command inputs.

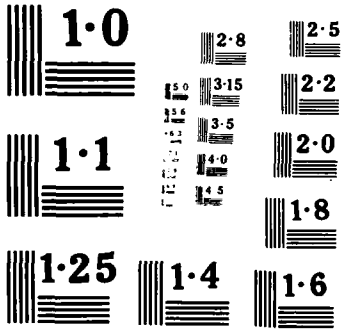
After a test scenario is loaded into the model system knowledge base, the simulation manager initializes the clock and scheduler and initiates the main control loop: the clock is advanced, timely events are injected into the simulator and prioritized; the scheduler allocates events, executes function behaviors, and determines successor events. If explicit component assignments have been made for functions (e.g., in the SRS), the scheduler automatically handles servers assignments, lockups and releases and event queueing.

It is important to note that ASDL models [illegible] directly: the simulation of function actions is [illegible] executing procedures attached to model system [illegible] explicit behaviors are ascribed directly to [illegible] systems and system components. Instead, [illegible] create, modify, or destroy such model elements. [illegible] standard simulators ignore functions, [illegible] with model system, component, and [illegible]

ASDL's approach was dictated by the [illegible] behaviors for acquisition stages when [illegible] system architectures and data [illegible] (SSS), in particular, deliberately [illegible]

AD-A189 819 ANNUAL KNOWLEDGE-BASED SOFTWARE ASSISTANT CONFERENCE (2ND) HELD ON 18-20 (U) ROME AIR DEVELOPMENT CENTER GRIFFISS AFB NY K M BENNER JAN 88 RADC-TR-87-243 4/4

UNCLASSIFIED F/G 12/5 NL


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5
5·0

Alternatively, detailed behaviors can be prescribed in the context of specific system architectures. ASDL test simulations have been written for portions of military communications and radar systems.

As the acquisition cycle progresses into design phases, system architectures and behaviors of subsystems and their components are defined in increasing detail. Function-oriented simulation becomes cumbersome and unintuitive. Accordingly, ASDL will either incorporate or interface with an object-oriented simulator tool to extend its applicability to later stages of acquisition.

ANALYSIS CAPABILITIES

ASDL's library elements include sizing and complexity factors that enter into early acquisition studies of economic feasibility (Figure 6). Users can construct system models, assign values to components attributes (e.g., sizing descriptors), and compute system sizing profiles that can be used as drivers to (external) costing tools such as COCOMO. ASDL's initial profiling utility totals estimated source lines and reusable lines of code from all software component instances in a system model. More sophisticated methods are being explored.

ASDL performs simple (i.e., syntactic), kinds of completeness testing. It can verify that functions are exhaustively allocated among system components (B-Spec/SRS) and that functional requirements (SSS) are accounted for (i.e., traced), in subsequent acquisition models (Source Selection, SRS, STLDD). Additional completeness checkers (e.g., for connectivity and decomposition relations), are being investigated.

ASDL utilities support users in specifying and simulating function flows and behaviors. Users can manually analyze the resulting flow structures, behaviors, and simulation traces to uncover omissions, ambiguities, and inconsistencies in behavioral aspects of system descriptions, (SSS Source Selection, SRS).

The simulator shell can be used to gather quantitative performance data as well as to study qualitative system behaviors. Each function behavior can be associated with either a numerical time interval or a user-specified functional expression which is evaluated in the simulation environment when a function behavior is actually executed. User "time expended" functions can reflect performance impacts due to system loads, server speeds, data object size, or other model system variables.

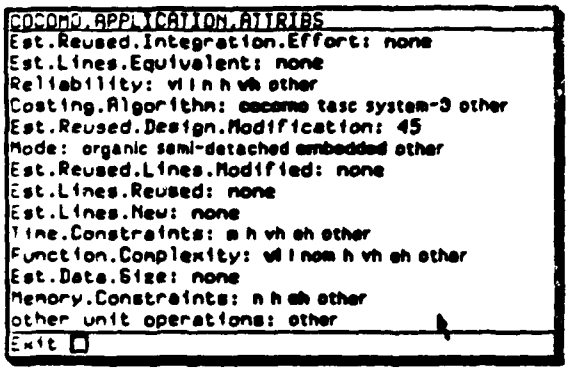


Figure 6. Component Sizing Attributes

USER INTERFACE

ASDL's intended user community is comprised of acquisition analysts and systems engineers. This audience is assumed to be familiar with systems and the acquisition process, but not with AI concepts or programming skills. Interface design is obviously critical to the tool's success: information and capabilities must be easily and intuitively specified and accessed, with a minimum of training.

Accordingly, the user interface depends heavily on graphics. Navigation through models and libraries is facilitated by tree-based representations (cf Figure 1). Menus guide users through activity selection, information specification and editing. Figure 6 displays the ASDL multiple-choice menu interface for information entry. (Figures 2 and 3 use KEE's frame editor to show all of the information actually contained in ASDL data structure at once.)

Menu-driven graphics editors are employed to specify relations: users are assisted in selecting and positioning of (labelled box) icons depicting model function and component classes, opening decomposition windows, and drawing and labeling (line) icons representing connectivity and function flow relations.

Items that cannot be specified graphically (e.g., constraints, branching and looping conditions in functional flows, and functional behaviors), are described declaratively. Users describe such data via intelligent menu-drivers coupled to dedicated languages. Language interpreters perform appropriate parsing, generation and installation of symbolic expressions or code (e.g., procedural attachments), and other bookkeeping measures.

The constraint language menu-drivers, display ingredients which users select to formulate antecedent and consequent clauses for conditional rules. Rule clauses describe predicate expressions involving model object classes or instances, their attributes and values, or simple (mathematical) functions of such values. Clauses can be combined using the standard logical connectives, as in the costing constraint example presented earlier. The interpreter parses the result, creates a constraint relation instance, installs the clauses on appropriate slots, and sets up demons that automatically check for constraint violations.

Similarly, menus based on ASDL's behavioral language enable users to specify data object creation and destruction and changes in component or data object attribute values. Menu selections for simple control structures (e.g., conditionals, loops, sorts), are also available to describe more complex function behaviors. Language enhancements will be based on user feedback and current experiments.

The scenario generator portion of the simulator interface is also a menu-driven editor. Scenario events are basically instances of class data objects or actions already defined in the model system description. Once the user identifies these classes, the generator presents multiple-choice menus (based on legal values facets) that guide users in specifying event attribute values. The generator also prompts the user for event injection time and functional flow entry point and adds bookkeeping slots that are user-transparent.

The primary simulator control loop includes an interrupt monitor that allows users to suspend simulations at any clock pulse and to access ASDL's model browsing utilities to investigate the current state of the system model and active data objects. The simulator shell also automatically maintains a graphic execution trace and logs of scenario runs to facilitate behavioral analysis.

FUTURE WORK

The utility of library-based modeling tools depends directly on the richness of their libraries: little, if any, gain in productivity is achieved if users must continually create new templates as they construct system descriptions. Accordingly, ASDL will be exercised using various acquisition programs in order to seed the generic library with a reasonable spectrum of C^3I function and component templates. As ASDL is used in specification development, the historical library will grow as well.

Currently, the generic modeling library depicts functions and components used in military communications systems, security requirements (from the DoD Trusted Computer System Evaluation Criteria, CSC-STD-003-85), an Air Defense Command and Control display system, and several radar systems.

The existing scenario editor only supports the creation of individual test events. It is clearly desirable, and relatively easy, to incorporate a batch event generator similar to standard discrete event simulators. Users will be able to create scenarios simply by specifying event types, population sizes, and distributions over attribute values (e.g., injection intervals, message field values).

ASDL's current analysis capabilities are admittedly somewhat limited, particularly with respect to automated analysis of simulations. The reason for this is that initial project efforts have focused on defining a basic representational framework and maintenance utilities. We are planning to investigate automated behavioral analysis capabilities and model post-processors in the near future.

Consider, for example, the problem of comparing simulation runs produced by behavioral models of a system across acquisition stages, driven by the same test scenario(s). One such comparison amounts to verifying that contractor specifications (e.g., SRS), satisfy behavioral aspects of customer requirements (e.g., SSS). Another verifies that a contractor specification (e.g., STLDD), refines rather than alters the behavioral aspects of preceding contractor products (e.g., SRS).

Intuitively, models from later phases of the development cycle are behaviorally complete and consistent just in case they produce the same kinds of events in the same sequences that were generated by simulations of earlier system models. Devising an algorithm to automate such a mapping across simulation event logs is a difficult problem.

First, earlier models (e.g., SSS), have a coarser event granularity than subsequent models (e.g. SRS): the former model describes function behaviors at a system level; the latter depicts behavior with respect to a particular system architecture, function allocation, and specification of data depicts and commands. Second, the combinatorics of event mappings are severe for detailed behavioral simulations (e.g., hundreds of events).

The functional approach to simulation taken by ASDL mitigates this problem somewhat since, by construction, the execution of a function on particular data objects is modeled in terms of a fixed, preprogrammed collection of events. Unfortunately, the remaining mapping, between functions in system models for different acquisition stages, can be a many-to-many relationship. ASDL currently lacks a semantics of functionality sufficiently robust to trace behaviors for such complex cases.

Future work will address this problem. It should also be noted that automated behavioral analyses such as the capability described here are simply not feasible without an environment like ASDL - most tools do not support simulation of system behavioral descriptions across a suitable range of acquisition phases.

The most challenging assignment is to make ASDL more intelligent. Currently, ASDL operates largely as a passive support tool: users perform all significant decisions, selecting appropriate templates, editing and connecting them, and describing behaviors. ASDL's current limitations stem from the fact that it only stores "whats" information - knowledge pertaining to systems and system ingredients, per se.

More active assistance capabilities require additional knowledge, the "hows" and "whys" of system engineering and of acquisition cycles: why components can or cannot be connected

together; how components implement functions; why some specifications are superior or inferior to others; and how one or more requirements entail additional "derived" requirements.

Our initial approach for resolving this problem will be to enhance the generic modeling library by establishing relational links between frame templates depicting generic functional areas, functions, and components. For example, the functional area Sensor Functions can be associated with specific missions, such as target detection, tracking, and classification. The functions resulting from a decomposition of this functional area (e.g., signal generation, transmission, reception, processing), can in turn be linked with behaviors or processes involving types of components (e.g., antennae, processors, displays). Individual functions (e.g., beam sweep), can be linked to specific classes of system components (e.g., scanners such as rotodomes, phased array antennae), and with sharply-defined design purposes (e.g., to obtain angular scan coverage).

In addition, an analogical reasoning capability needs to be developed. Enhanced in this way, ASDL should be able to extract and combine library elements to suggest partial functional decompositions to users based on high-level descriptions of requirements (e.g., fault-tolerant message processing). We expect the addition of "how" and "whys" system engineering expertise to ASDL knowledge bases will also facilitate the solution of the behavioral analysis problem described above.

Another important avenue of inquiry is the integration of tools like ASDL into the existing DoD process. In particular, it is desirable to develop a post-processor capable of translation ASDL system models into rough drafts of DoD (2167) standard documentation products. However, given current natural technology limitations and the fact that DoD specifications are legally binding documents, additional manual processing will clearly be necessary.

The second important integration problem to be addressed is to interface ASDL, which is essentially an acquisition process front-end tool to back-end tools (i.e., for detailed design and software implementation.) Example back-end tools include (intelligent) programming environments (e.g., dedicated to Ada and Ada PDL code generation and listing), and automated programming systems. This work is an important prerequisite to developing a fully integrated intelligent environment to support users through the complete acquisition process.

SUMMARY

ASDL is a prototype AI-based tool for acquisition support. The objective of ASDL is to assist analysts and system engineers in generating and analyzing symbolic models of functional requirements and contractor system development products through high level design. "Knowledge" is distributed across a variety of structures in ASDL: the knowledge representation substructure (frames, and procedural attachments); C^3I expertise encoded in modeling library templates; specification and design information in system model knowledge bases; the simulator shell; and user interface and analysis utilities.

REFERENCES

1. Defense System Software Development, June 4, 1985, Bureau of Standards, AMSC No. N3608.

2. Borgida, A., Greenspan S., and Mylopoulos J. (1985), Knowledge Representation as the Basis for Requirements Specifications, IEEE Computer, Vol. 18, No. 4, pp. 82-91.

3. Fikes, R. and Kehler, T. (1985), The Role of Frame-Based Representation in Reasoning, Communications of the ACM, Vol. 28, No. 9, pp. 904-920.

4. Kunz, J., Kehler, T. and Williams, M. (1984), Applications Development Using a Hybrid AI Development System, AI Magazine, Vol. 5, No. 3, pp. 41-54.

5. Lenat, D., Prakash, M. and Shepherd, M. (1986), CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks, AI Magazine, Vol. 6, No. 4, pp. 65-85.

6. Stefik, M., and Bobrow, D. (1986), Object-Oriented Programming: Themes and Variations, AI Magazine, Vol. 6, No. 4, pp. 40-62.

7. Stefik, M., Bobrow, D., Mittal, S., and Conway, L. (1984), Knowledge Programming in LOOPS: Report on an Experimental Course, AI Magazine, Vol. 4, No. 3, pp. 3-12.

8. Hollan, J., Hutchins, E. and Weitzman, L. (1984) STEAMER: An Interactive Inspectable Simulation-Based Training System, AI Magazine, Vol. 5, No. 2, pp. 15-27.

Acknowledgment: We wish to thank the Program Managers of the Computer Resource Management Technology Program (PE64740F), Electronic Systems Division, U. S. Air Force, for their assistance and guidance in this project. We would also like to acknowledge the assistance of Dr. M. J. Prelle at MITRE and Captain J. P. Dean at ESD/ACCR.

The work described in this paper is the result of a collaborative effort that includes Judith A. Marcet and Stuart Goldkind. We would like to thank these project members for their important contributions to ASDL system design and implementation, particularly in the areas of user interface, constraints and relations.

Plan Recognition in GRAPPLE: An Intelligent Assistant for the Process of Programming

Karen E. Huff
Victor R. Lesser

Computer and Information Science Department
University of Massachusetts
Amherst, MA. 01003

Abstract: We are prototyping a system called GRAPPLE that provides intelligent assistance to the programmer carrying out the *process* of programming. The architecture, based on an AI *planning* paradigm, can provide both passive and active assistance. Passive assistance, accomplished by *plan recognition*, is used to detect and avert process errors. Active assistance, accomplished by *planning*, is used to automate the programming process. We will be demonstrating the plan recognition facilities, and the programmer assistance based upon plan recognition, in our current prototype running on a Texas Instruments Explorer.

In GRAPPLE, software processes are defined in a hierarchical, state-based plan formalism that has been engineered to meet the demands of complex domains. The plan recognizer has the ability to recognize *partial* plans in this formalism, focusing on a best interpretation when there is not yet enough information to disambiguate competing interpretations. It can also handle interleaved actions from multiple, top-level plans. The types of errors that can be detected include attempts to perform an action whose precondition is not met, or one that has no interpretation, or one that would disrupt a previously satisfied precondition for another expected action. We will show the recognition of some simple software process plans, with emphasis on the algorithms and data structures internal to the plan recognizer.

Demonstration of the Knowledge-Based Specification Assistant[1]

W. Lewis Johnson
USC/Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292

Abstract

This demonstration will provide an overview of the Knowledge-Based Specification Assistant system, one of the Knowledge-Based Software Assistant projects being funded by RADC. This system assists specifiers in developing specifications, evaluates specifications, and explains the contents and development of specifications. Aspects of each of these three kinds of capabilities will be shown.

[1]This research is supported by the Air Force Command, Rome Air Development Center under Contract No. F30602-85-C-0221. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of RADC, the U.S. Government, or any person or agency connected with them.

1. Overview

If software development were specification based, a number of benefits would result. By directly executing the specification before implementation, errors would be found more quickly. Since a specification states explicitly what a system should do, rather than how it should do it, a specification is usually easier to understand and maintain than an implementation code. At the same time, specification-based software development poses some challenges. Specifications of complex systems are themselves complex; people reading specifications can get lost in the details. Furthermore, it takes a lot of work to build a specification; one must identify what requirements must be met, and decide what tradeoffs to make in meeting those requirements. In the Knowledge-Based Specification Assistant project, we attempt to meet these challenges by helping people to develop specifications, to evaluate specifications, and to explain specifications.

The Specification Assistant appears to the user as an extended structure editor. As in a display-oriented structure editor, the specification is presented to the user as text, but the user can manipulate its structure. But whereas a structure editor would simply represent the specification's syntax, we represent both its syntax and semantics. This allows for more powerful cooperation between user and system. We have implemented these editing facilities as an extension of Symbolics's ZMACS editor.

Figure 1-1 shows how the system presents itself to the user. There is a window in which the specification text that is currently being worked on appears. To the right of this window is an area for menus. There is separate menu pane for each class of operations that the Specification Assistant can perform: e.g. structure-editing operations, symbolic evaluation operations, and specification refinement transformations.

The specification shown in the figure is part of the specification for an air traffic control system. The specification is written in GIST, a specification language developed at ISI [1]. The specification defines some of the entities in the air-traffic-control domain, such as air spaces, controllers, and radar tracks, and some of the relations between them such as the `controls` relation that holds between controllers and flights

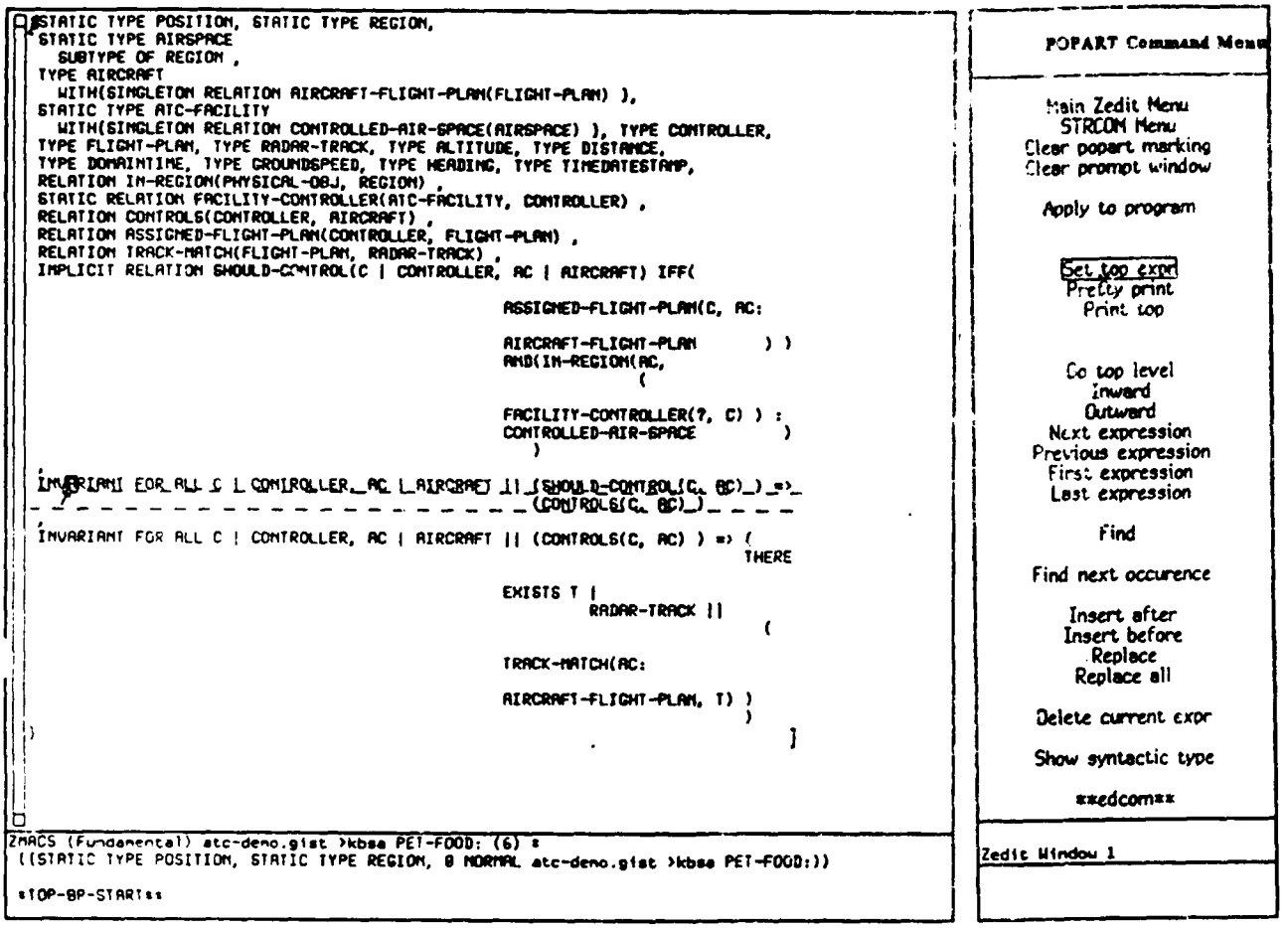


Figure 1-1: Structure-Editing Facilities in KBSA

that they control. The specification also contains a some invariants, i.e., facts which must hold true of the specified system at all times. In the menu area we see the structure-editing commands of the POPART language manipulation system [2]. POPART is one of the general-purpose tools built at ISI that have been used in constructing the Specification Assistant. The structure-editing commands include commands such as "Inward" and "Next expression" which move the editor focus to different parts of the expression, and commands like "Insert after" and "Replace" which change modify the specification at the current point of focus.

2. Specification Assistance Facilities

The Knowledge-Based Specification Assistant assists the software process in three ways:

- it helps the specifier construct the specification;
- it evaluates the specification, showing what behavior it describes;
- it explains the specification and its development, so that specifiers and non-specifiers alike can understand what the specification says.

Each of these facilities will be shown in the demonstration.

The Specification Assistant provides semantics-oriented specification transformations, which we call *high-level editing commands*, for refining and elaborating specifications. A history of the specification development is kept, so that a user can compare earlier, abstract versions of the specification with later, detailed versions. High-level editing commands allow specifiers to first state what system functionality would ideally be desirable, and then refine that into something that is actually achievable. For example, one of the ideal desirables stated in the specification in Figure 1-1 is

```
INVARIANT FOR ALL c|controller, ac|aircraft ||
  (should-control(c, ac)) => (controls(c, ac)),
```

meaning that every aircraft that a controller should control is in fact controlled. In Figure 2-1, we see the specification after a refinement has taken place. The editing command that was applied was "Maintain Invariant Reactively". The invariant is now gone; instead, there is a demon which reacts to the presence of uncontrolled flights and

attempts to make then controlled. Although the invariant is gone, it is still present in the history. As a consequence, a specifier can always look back to find out what high-level goals some complex specification is attempting to maintain.

KBSA provides two evaluation tools: a symbolic evaluator, and a simulator. The symbolic evaluator is designed to allow the user to execute the specification on abstract symbolic data. It attempts to prove general theorems about how the specification operates on arbitrary data. For example, the user can tell the symbolic evaluator to assume that there is some arbitrary number of aircraft and controllers in the system, and have it try to prove that all aircraft in the air space are controlled. If the symbolic evaluator cannot prove it, there may be an error in the specification.

The simulator allows the user to run concrete cases through the specification, and observe the result. It can handle more complex test cases than the symbolic evaluator can, but it cannot prove anything about how the specification will behave in general. One important feature is its ability to simulate the behavior of a number of processes executing simultaneously.

Our basic means for explaining a specification is to generate an English paraphrase for different parts of the specification. We are now building a general-purpose explanation shell which can field user queries and plan explanations either in English or in some other presentation medium. The English paraphraser will be placed under control of this explanation engine. As a result, the paraphrases will be responsive to the user's interests; formerly the paraphraser would paraphrase the entire specification. We ultimately envision extending the explanation facility so that maintainers and clients can ask questions about the system and its development, and so that specifiers can even ask questions about how to use the Specification Assistant's facilities.

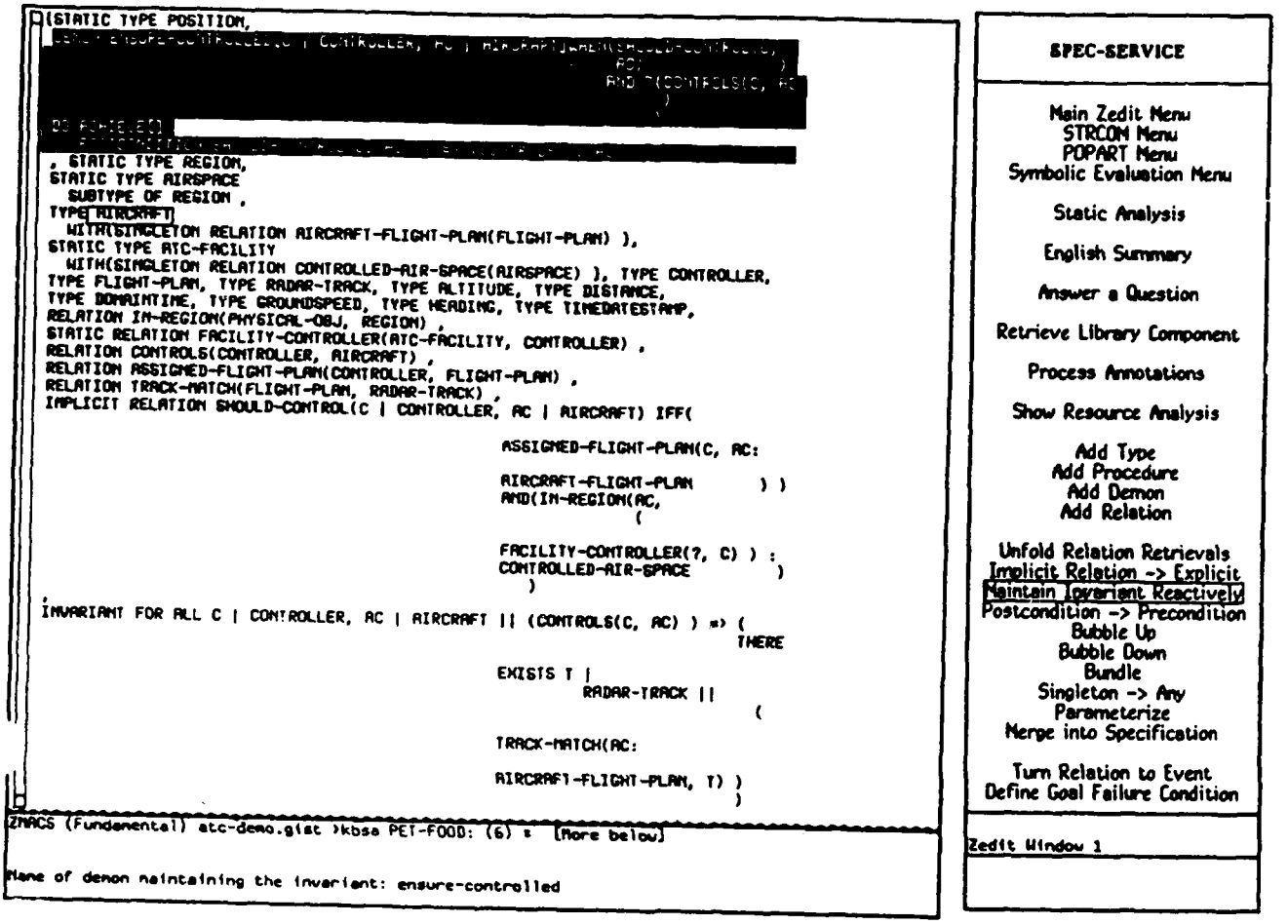


Figure 2-1: A Refined Version of the Air-Traffic Control Specification

References

1. Goldman, N. and D. Wile. Gist Language Description. Draft.

2. Wile, D. *POPART: Producer of Parsers and Related Tools. System Builders' Manual.* USC Information Sciences Institute, 1981.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.

END
DATE
FILM
4-88
DTIC