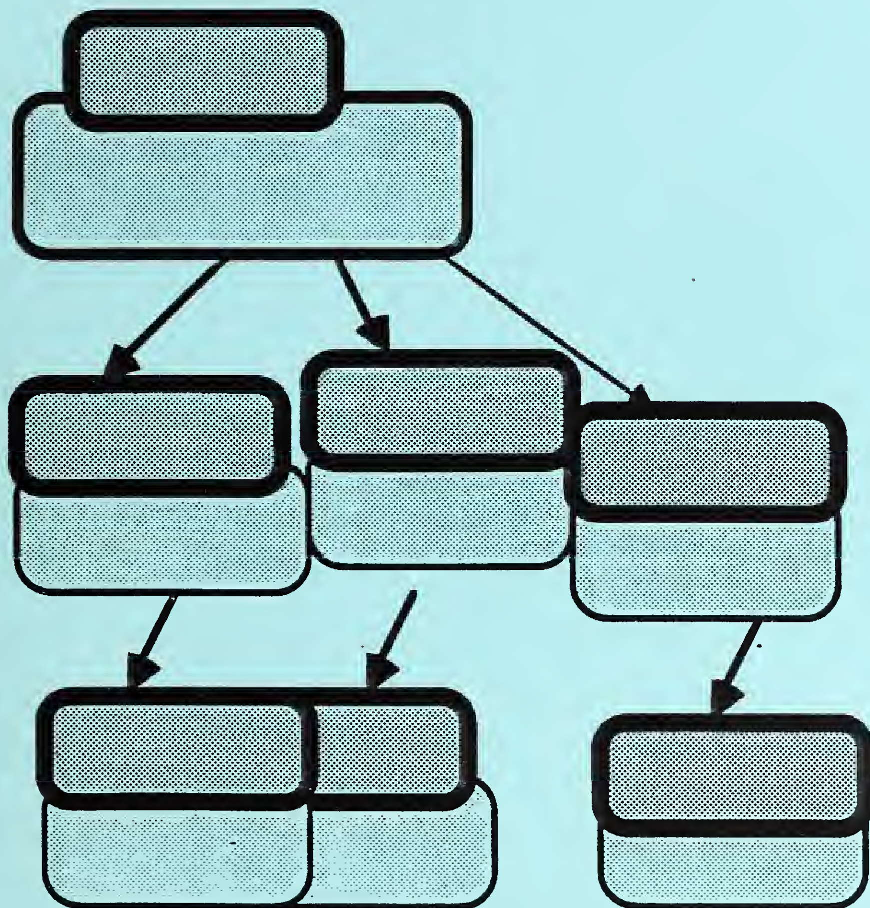
NAT'L INST. OF STAND & TECH R.I.C.
A11103 462112

NISTIR 4503

NIST
PUBLICATIONS

Proceedings of the Object-Oriented Database Task Group Workshop Tuesday, May 22, 1990 Atlantic City, NJ



Edited by:
Elizabeth N. Fong

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Computer Systems Laboratory
Gaithersburg, MD 20899

Craig W. Thompson

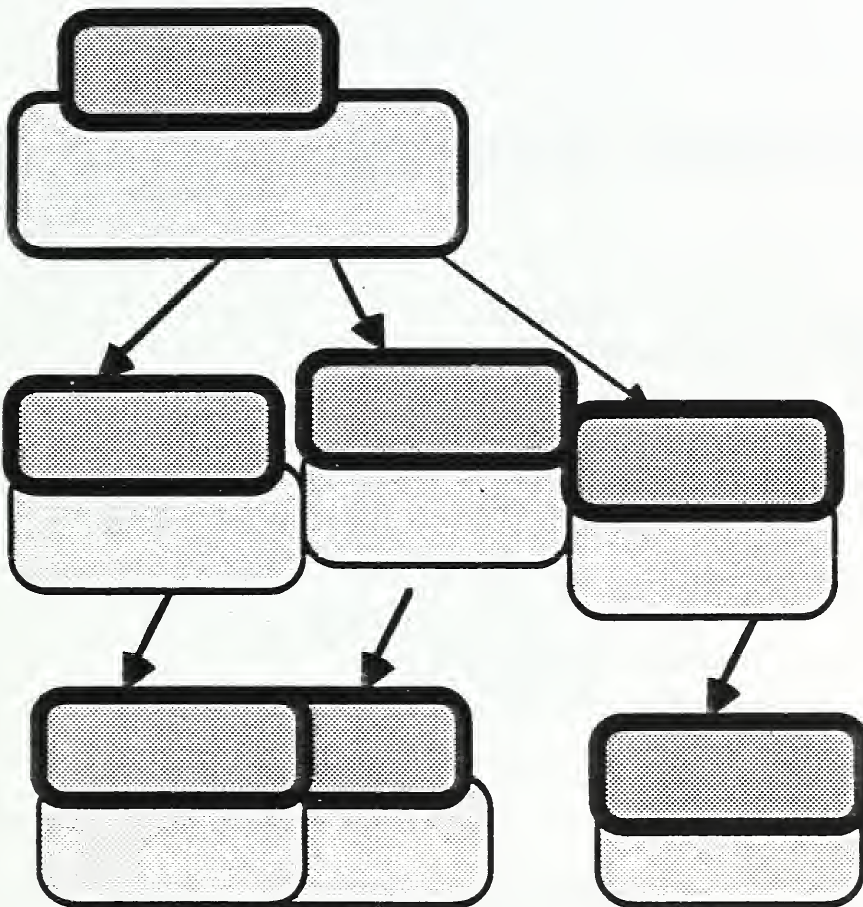
Texas Instruments Incorporated
Dallas, TX 75265

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

QC
100
.U56
#4503
1991
C.2

NISTIR 4503

Proceedings of the Object-Oriented Database Task Group Workshop Tuesday, May 22, 1990 Atlantic City, NJ



Edited by:
Elizabeth N. Fong

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Computer Systems Laboratory
Gaithersburg, MD 20899

Craig W. Thompson

Texas Instruments Incorporated
Dallas, TX 75265

February 1991

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

PROCEEDING OF THE

FIRST OODB STANDARDIZATION WORKSHOP

X3/SPARC/DBSSG/OODBTG

Atlantic City, New Jersey
May 22, 1990

Edited by:
Elizabeth Fong
Craig W. Thompson

Program Committee:
Gordon Everest, University of Minnesota
Elizabeth Fong, National Institute of Standards and Technology
Bill Kent, Hewlett-Packard Laboratories
Haim Kilov, Bellcore
Ken Moore, Digital Equipment Corporation
Allen Otis, Servio Corporation
Mark Sastry, Honeywell
Craig Thompson, Texas Instruments Incorporated

ABSTRACT

This report constitutes the proceedings of a one-day workshop on standardization of object database systems held in Atlantic City, New Jersey, on May 22, 1990. The workshop was sponsored by the Object-Oriented Database Task Group (OODBTG) of the ASC/X3/SPARC Database Systems Study Group (DBSSG).

This workshop, held the day before the ACM International Conference on Management of Data (SIGMOD '90 conference), was the first of two workshops held to solicit public input to identify what aspects of object database systems may be candidates for consensus that can lead to standards. The second companion workshop was held on October 23, 1990, in Ottawa, Canada, coincident with the Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA).

The workshop attempted to focus on concrete proposals for language or module interfaces, exchange mechanisms, abstract specifications, common libraries, or benchmarks. The workshop announcement also solicited papers on the relationship of object database system capabilities to existing standards, including assertions that question the wisdom of standardization.

This proceedings consists of 22 position papers covering various aspects where standardization on object database systems may be possible.

NIST is publishing the proceedings of both of these workshops to disseminate information on object standardization activities. The proceedings of the second workshop on standardization of object database systems appeared as NISTIR 4488.

Key words: Database; database management system; DBMS; data model; object-oriented; OODB; programming languages; standards.

DISCLAIMER

The views expressed in this report are those of the authors, and do not necessarily reflect the views of the National Institute of Standards and Technology (NIST) or any of its staff. The specific vendors and commercial products identified in this report do not imply recommendation or endorsement by the NIST.

The report has not been subject to policy review or direction by the NIST, nor by Accredited Standards Committee X3 Information Processing Systems, Standards Planning and Requirements Committee (SPARC).

FOREWORD

Object-oriented database systems (OODBs) have reached the point where it makes sense to consider their potential for formal standardization. There are now several OODB products and substantial research prototypes. Several authors have stated requirements for OODBs and have offered definitions and initial specifications for consideration.

In January, 1989, the Database Systems Study Group (DBSSG), one of the advisory groups to the Accredited Standards Committee X3 (ASC/X3), Standards Planning and Requirements Committee (SPARC), operating under the procedures of the American National Standards Institute (ANSI), established a task group on Object-Oriented Databases (OODBTG). To facilitate further development and use of OODB technology, OODBTG seeks:

- To define a common reference model for an Object-Oriented Database, based on object-oriented programming and database management system models.
- To assess whether and where standardization on OODBs is possible and useful. Some areas of possible standardization include glossary, reference model, operational model, interfaces, and data exchange.
- To complete a Final Report in 1991 containing recommendations regarding future ASC/X3 standards activities in the object-oriented database area, including how these standards would relate to existing standards.

OODBTG meets quarterly. Persons interested in OODBTG should contact Elizabeth Fong, National Institute of Standards and Technology, Building 225, Room A266, Gaithersburg, MD 20899 (301-975-3250).

The purpose of the May 22 OODBTG Workshop is to identify areas where consensus on OODBs may be possible and desirable in a setting where authors can expect feedback and possible action on their ideas. This workshop solicits public feedback from the database community. A companion workshop planned for October 23, 1990, at OOPSLA'90 will canvas the programming language community.

The Call for Participation attracted 22 papers, all of which appear in the proceedings. The program committee selected 14 papers for presentation, based on relevance to OODBTG's mission. Fifty-five people participated in the workshop.

The papers reprinted in this volume represent the opinions of the individual authors. These papers are neither approved standards nor recommendations of OODBTG. Neither OODBTG nor the Program Committee have made any

judgements as to whether any topic of any paper complies with the Reference Model currently under development by OODBTG.

As evidence of the wide-spread interest in OODB standardization, we received papers and preregistrations from:

Australia, Canada, England, France, Japan, Republic of Korea, Netherlands, USA, W. Germany

Object Design, Objectivity, Object Sciences, Ontologic, Altair O2, Servio Corporation, Oracle, EIS, CAD Framework Initiative, OSF, Mentor, Ashton-Tate, Concurrent Computer Corp, DEC, Xerox, Honeywell, Texas Instruments, Hewlett-Packard, McDonnell-Douglas, Boeing, Tandem, Grumman, NEC, Nixdorf, Bell Communications Research, MCC, DARPA, CECOM, NIST, NOSC, U Toronto, Kyoto U, U Michigan, U Maryland, UT Austin, U Wisconsin, Stanford, Columbia, Rensselaer U, U Alberta, Australian National University at Canberra, AT&T, UC Berkeley, IBM Research, Tandem Computers, University of Wisconsin, DEC, and Oracle Corporation, Summa International

The workshop proceedings begins with background material on OODBTG, then shifts to selected presentations. The workshop itself ends with a Feedback Panel and is followed by a Birds of a Feather session to be held during Sigmod. At the workshop conclusion, participants will be asked to complete feedback forms to indicate areas where they think consensus may be possible, areas where roadblocks might block consensus, and comments on the OODBTG ODB Reference Model. The results of the workshop are inputs to OODBTG which affect the OODBTG ODB Reference Model and the OODBTG Final Report.

I express my thanks to all authors who submitted papers and to the members of the program committee who reviewed them. Special thanks goes to Elizabeth Fong, NIST, who co-edited the proceedings.

Craig Thompson
Workshop Chairman

Table of Contents

PAGE

WORKSHOP ATTENDEES LIST 5

OODBTG STATUS

"Workshop Objecti 11
OODBTG Workshop Chairman, Craig Thompson, Texas Instruments Incorporated

"The Role of Standards" 15
X3/SPARC/DBSSG Chairman, Ed Stull, Summa International

"OODBTG Charter, Progress, and Plan" 24
OODBTG Chairman, Tim Andrews, Ontologic

"OODB Reference Model (Draft)" 30
OODBTG, presented by Allen Otis, Servio Corporation

REQUIREMENTS, REFERENCE MODELS

"Object-Oriented DBMS Requirements" 60
Keith A. Marrs and Laila G. Robinson, McDonnell-Douglas Corporation

"Third Generation Data Base System Manifesto" 68
Michael Stonebraker, Lawrence Rowe, Bruce Lindsay, James Gray, Michael Carey, Philip Bernstein, and David Beech, {UC Berkeley, UC Berkeley, IBM Research, Tandem Computers, University of Wisconsin, DEC, and Oracle Corporation, respectively}

TAXONOMY OF STANDARDS

"ANSI OODBTG Workshop Position Paper" 84
Leon Guzenda and Andrew Wade, Objectivity, Inc.

OO DATA MODELS

"The Object Standardization Challenge" 94
Bill Kent, Hewlett Packard Corporation

"Application Object Model for Engineering Information Systems" 100
Jonathan W. Krueger, Honeywell Systems and Research Center

PERSISTENT LANGUAGE

"Principles for Persistent Object Access" 110
Fred Loney, Mentor Graphics Corporation

"Notes toward a Standard Object-Oriented DDL and DML" 116
Thomas Atwood and Jack Ornstein, Object Design Inc.

Table of Contents (continued)

PAGE

OBJECT QUERY LANGUAGE

"A Model for OODB Queries" 126
David D. Straube and M. Tamer Ozsu, The University of Alberta, Edmonton, Alberta, Canada.

"Intelligent SQL" 136
Setrag Khoshafian, presented by Razmik Abnous, AshtonTate

"Strawman Reference Model for Object Query Language" 166
Jose' Blakeley, Craig Thompson, Abdallah Alashqur, Texas Instruments, Incorporated

"Important Features of Iris OSQL" 180
Bill Kent, Hewlett Packard Corporation

CHANGE MANAGEMENT

"Strawman Reference Model for Change Management of Objects" 190
John Joseph, Mark Shadowens, John Chen, and Craig Thompson, Texas Instruments Incorporated

OO INTEROPERABILITY FRAMEWORK

"EIS/XAIT Project: An Object-based Interoperability Framework for Heterogeneous Systems" 212
Girish Pathak, Bill Stackhouse, and Sandra Heiler, Xerox Advanced Information Technology, Cambridge, MA.

"Goals and Requirements, Storage Manager (SM) Working Group" 222
Andrew Wade (editor and chairman SMWG/CFI), Design Data Management TSC, CAD Framework Initiative, presented by Jason Browning, AT&T

FEEDBACK PANEL, moderator: Allen Otis, Servio Corporation
- -Tim Andrews, Ontologic
- -Mike Carey, U Wisconsin
- -Bill Kent, Hewlett Packard
- -Tom Atwood, Object Design

X3/OODBTG BIRDS-OF-A-FEATHER MEETING (during SIGMOD)
- -OODBTG's OODB Reference Model
- -Taxonomy of Potential OODB Standards
- -Issues, Roadblocks

Table of Contents (continued)

	PAGE
PAPERS NOT PRESENTED AT THE WORKSHOP	
"Object-Oriented Data Modelling in Rule-Based Software Development Environments" Naser S. Barghouti and Michael H. Sokolsky, Columbia University	*234*
"An Entity-Oriented Data Model--MIX" Tzy-Hey Chang, Digital Equipment Corporation	*242*
"Object Data Model = Object-Oriented + Semantic Models" Qing Li, Australian National University, Canberra	*258*
"Towards an Optimum Language Data Model" Ed Lowry, Digital Equipment Corporation	*266*
"A Neutral Object-Oriented Data Model" Robert Marcus, Boeing Advanced Technology Center	*274*
"OODB Standardization" Roger Osborn, Concurrent Computer Corporation Ltd.	*276*
"Standardization of Object-Oriented Database Systems" Daniel O. Sanderson, Digital Equipment Corporation	*286*
"OODBTG Workshop on Standards Position Paper" Donald Sanderson, Rensselaer Polytechnic Institute	*294*

OODB Workshop Attendees

Razmik Abnous
Ashton-Tate
2033 N. Main #980
Walnut Creek, CA 94596
Tel: 415-746-3262
Fax: 415-746-1559
Email: abnous@alexis.a-t.com

Mark Anderson
Texas Instruments
PO Box 655012 M/S 3635
Dallas, TX 75265
Tel: (214) 917-2210
Email: anderson@ticipa.ticom,
anderson%ANDM@timsg.cs.ti.com

Tim Andrews
Ontologic, Inc.
Three Burlington Woods
Burlington, MA 01803
Tel: 617-270-9797
Email: uunet!ontologic!andrews

Thomas Atwood
Object Design, Inc.
One New England Executive Park
Burlington, MA 01803
Tel: 617-270-9797
Email: tom@odi.com

Naser S. Barghouti
Department of Computer Science
Columbia University
New York, NY 10027
Tel: 212-854-8182
Email: naser@cs.columbia.edu

Doug Barry
Itasca Systems, Inc.
2850 Metro Drive, Suite 300
Minneapolis, MN 55425
tel: 612-851-3155
fax: 612-854-4834

Don Batory
Department of computer Science
The University of Texas at Austin
Austin, TX 78712
Tel: 512-471-9711
Fax: 512-471-8885
Email: dsb@cs.utexas.edu

David Beech
Oracle Corporation
20 Davis Drive
Belmont, CA 94002
Tel: 415-358-3457
Fax: 415-349-6374
Email: dbeech@us.oracle.com

Jose' Blakeley
Texas Instruments Incorporated
PO Box 655474, MS 238
Dallas, TX 75265
Tel: 214-995-0328
Fax: 214-995-0304
Email: blakeley@csc.ti.com

Mike Carey
Computer Sciences Department
University of Wisconsin
1210 West Dayton St.
Madison, WI 53706
Tel: 608-262-2252
Fax: 608-262-9777
Email: carey@cs.wisc.edu

Leon Cejas
67 E 2nd St. #7
New York, NY 10003
Tel: 212-529-8674

Tzy-Hey Chang
Applied Intelligent System Group
Digital Equipment Corporation
5 Scottswood Dr.
Sudbury, MA 01776
Tel: 508-490-8926 (or 8826?)
Email: chang@aisg.enet.dec.com

J. Charles Crabb
OCLC
MC 610
6565 Frantz Rd.
Dublin, OH 43017-0702
Tel: 614-761-5118
Email: jcc@rsch.oclc.org

Vineeta Darnis
GIP Altair
Domaine Voluceau BP 105
Le Chesnay cedex 78153
FRANCE
Fax: (33) 1 39 63 58 88 or 90
Email: vineeta@bdblues.altair.fr

OODB Workshop Attendees

Linda Dodge
Shell Development Company
PO Box 481
Houston, TX 77001
Tel: 713-663-2196
Email: linda@shell.com

Andrew J. Eisenberg
Digital Equipment Corp
55 Northeastern Blvd NU01-1/B09
Nashua, NH 03062
Tel: 508-884-1857
Fax: 508-884-0829
Email: eisenberg@sqlrus.enet.dec.com,decwrl!sqlrus.enet!eisenberg

Elizabeth Fong
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899
Tel: 301-975-3250
Fax: 301-590-0932
Email: fong@ise.ncsl.nist.gov

Juergen Friedrich
Nixdorf Computer AG / CADLAB
Bahnhofstrasse 32
D-4790 Paderborn
West Germany
Tel: (49) 52 51 284 128
Fax: (49) 52 51 284 140
Email: frido@cadlab.de,
frido@cadlab.uucp

John Joseph
Texas Instruments Incorporated
PO Box 655474, MS 238
Dallas, TX 75265
Tel: 214-995-0305
Fax: 214-995-0304
Email: joseph@csc.ti.com

Staffan Karlquist
Swedish Embassy
600 New Hampshire Ave NW
Washington D.C. 20037
Tel: 202-337-5187
Fax: 202-337-6108

Bill Kent
Hewlett Packard Laboratories
1501 Page Mill Road
P. O. Box 10490
Palo Alto, CA 94303-0969
Tel: 415-857-8723
Fax: 415-852-8137
Email: kent@hplabs.hp.com

Setrag Khoshafian
Ashton-Tate
2033 N. Main #980
Walnut Creek, CA 94596
Tel: 415-746-1550

Haim Kilov
Bell Communications Research
MRE 1E243
435 South Street
Morristown, NJ 07960
Tel: 201-829-2816
Fax: 201-292-0477

Kazuya Koda
Manager System Software
Fujitsu America, Inc.
Email: kazuk@fai.com

Jonathan W. Krueger
Honeywell SRC
3660 Technology Dr.
M/S MN65-2100
Minneapolis, MN 55418
Tel: 612-782-7642
Fax: 612-782-7438
Email: kreuger@src.honeywell.com

Krishna G. Kulkarni
Digital Equipment Corp.
1175 Chapel Hills Drive
Colorado Springs, CO 80919
Tel: 719-260-2718

Misook Lim
Human Computers, Inc.
Samyoung B/D
840 Yoksam-dong Kangnam-ku
Seoul 135-080
Republic of Korea
Tel: +82-2-553-0818
Fax: +82-2-553-0817
Email: jhhur@cosmos.kaist.ac.kr

OODB Workshop Attendees

Qing Li
Department of Computer Science
Australian National University
Canberra, ACT 2601, Australia
Tel: (062)-49-3783
Fax: (062)-49-0010
Email: qing@anucsd.anu.oz.au

Ken Moore
Digital Equipment Corporation
Mail Stop ZK02-1/N20
110 Spit Brook Road
Nashua, NH 03062-2698
Tel: 603-881-0547
Fax: 603-881-0120

Penny Muncaster-Jewell
McDonnell Douglas Space Systems
MS TB218
16055 Space Center Blvd
Houston, TX 77062-6208
Tel: 713-283-4350
Fax: 713-283-4020
Email: penny@mpad.span.nasa.gov

Patrick O'Connor
Data General Corp
4400 Computer Drive
Westobor, MA 01580
Tel: 508-870-6911
Fax: 508-898-2785
Email:
"Patrick_O~-Connor@oa.ceo.dg.com"

Jack Ornstein
Object Design, Inc.
One New England Executive Park
Burlington, MA 01803
Tel: 617-270-9797
Email: jack@odi.com

Roger Osborn
European Software Development Group
Concurrent Computer Corp Ltd.
227 Bath Rd.
Slough, Berkshire SL1 4AX
ENGLAND
Tel: 44-753-34511
Fax: 44-753-71661
Email: ro@concurrent.co.uk

Allen Otis
Servio Corporation
15220 NW Greenbrier Pkwy
Suite 100
Beaverton, OR 97006
Tel: 503-629-8383
Fax: 503-629-8556
Email:
uunet!servio!otisa,otisa@slc.com

M. Tamer Ozsu
Department of Computing Science
815 General Services Bldg.
The University of Alberta
Edmonton, Alberta
CANADA
Tel: 403-429-2860
Fax: 403-429-1071
Email: oszu@cs.ualberta.ca

Girish C. Pathak
Xerox Advanced Information Technology
4 Cambridge Center, 4th Floor
Cambridge, MA 02142
Tel: 617-499-4498
Fax: 617-499-4409
Email: pathak@xait.Xerox.COM

Ravi S. Raman
Bell Atlantic Knowledge Systems
9 South High St.
Morgantown, WV 26505
Tel: 304-291-2651 (BAKS), 304-293-7226 (CERC)
Fax:304-284-1391 (BAKS), 304-293-7541 (CERC)
Email:rsr@cerc.wvu.wvnet.edu

Laila G. Robinson
McDonnell Douglas Corporation
Dept 469, MC 0861020
PO Box 516
St. Louis, Missouri 63166
Tel: 314-298-4653
Fax: 314-298-4624

Daniel O. Sanderson
Digital Equipment Corporation
1175 Chapel Hills Drive
Colorado Springs, CO 80920
Tel: 712-260-2795 (or 217-260-2730)
Email: sanderson@cookie.dec.com

OODB Workshop Attendees

Donald Sanderson
Design Research Center
Rensselaer Polytechnic Institute
Bldg CII Room 7015
Troy, NY 12180-3590
Tel: 518-276-6751
Fax: 518-276-2702
Email: sandersn@rdrc.rpi.edu

Chander Sarna
Chips and Technologies
3050 Zanker Rd.
San Jose, CA 95134
Tel: 408-434-0600
Email: chander@chips.com

Michael H. Sokolsky
Department of Computer Science
Columbia University
New York, NY 10027
Tel: 212-854-8348
Email: sokolsky@cs.columbia.edu

Ellen M. Staelin
Spang Robinson Wiley Publications
PO Box 82228
Wellesley, MA 02181
Tel: 617-235-3631

Ed Stull
DBSSG chair
Summa International
13241 Osterport Dr.
Silver Spring, MD 20906
Tel: 301-942-4355

Mary-Ellen Stull
Summa International
13241 Osterport Dr.
Silver Spring, MD 20906
Tel: 301-942-4355

Satish M. Thatte
Texas Instruments Incorporated
PO Box 655474, MS 238
Dallas, TX 75265
Tel: 214-995-0340
Fax: 214-995-0304
Email: thatte@csc.ti.com

Peter Thiesen
XIDAK, Inc.
3475 Deer Creek Road, Bldg. C
Palo Alto, CA 94304
Tel: 415-855-9271
Fax: 415-855-9005
Email: peter@xidak.com

Craig Thompson
Texas Instruments Incorporated
PO Box 655474, MS 238
Dallas, TX 75265
Tel: 214-995-0347
Fax: 214-995-0304
Email: thompson@csc.ti.com

Stephen Turczyn
US Army CECOM
Center for Software Engineering
Attn: AMSEL-RP-SE-AST-SS (S. Turczyn)
Bldg 1210
Fort Monmouth, NJ 07703-5000
Tel: 918-532-2672
Email: turczyns@ajpo.sei.cmu.edu

Eugene N. Vasilescu
Grumman Data Systems
100 Woodbury Rd.
Woodbury, NY 11797
MS D12-237
Email: vasilesc@ajpo.sei.cmu.edu

Andrew E. Wade
Objectivity, Inc.
800 El Camino Real, 4th Floor
Menlo Park, CA 94025
Tel: 415-688-8000
Fax: 415-325-0939
Email: drew@objy.com

Charles Wan
David Sarnoff Research Center
Princeton, NJ
Tel: 609-734-2596
Fax: 609-734-2313
Email: csw@sarnoff.com

OODB Workshop Attendees

Ouri Wolfson
Computer Science Department
Columbia University
New York, NY 10027
Fax: 212-666-0140
Email: wolfson@cs.columbia.edu

Masatoshi Yoshikawa
Kyoto Sangyo University
Dept of Info and Communication
Sciences
Kyoto 603
JAPAN
Tel: +81(75)701-2151
Fax: +81(75)722-3034
Email: yosikawa@kyoto-su.ac.jp

Workshop Objective

Workshop Chairman Craig Thompson,
Texas Instruments Incorporated

WORKSHOP OBJECTIVE

The goal of this one-day workshop is to solicit public input from the database community and to identify what aspects of OODBs may be candidates for consensus that can lead to standards. A companion workshop planned for OOPSLA'90 will canvas the programming language community.

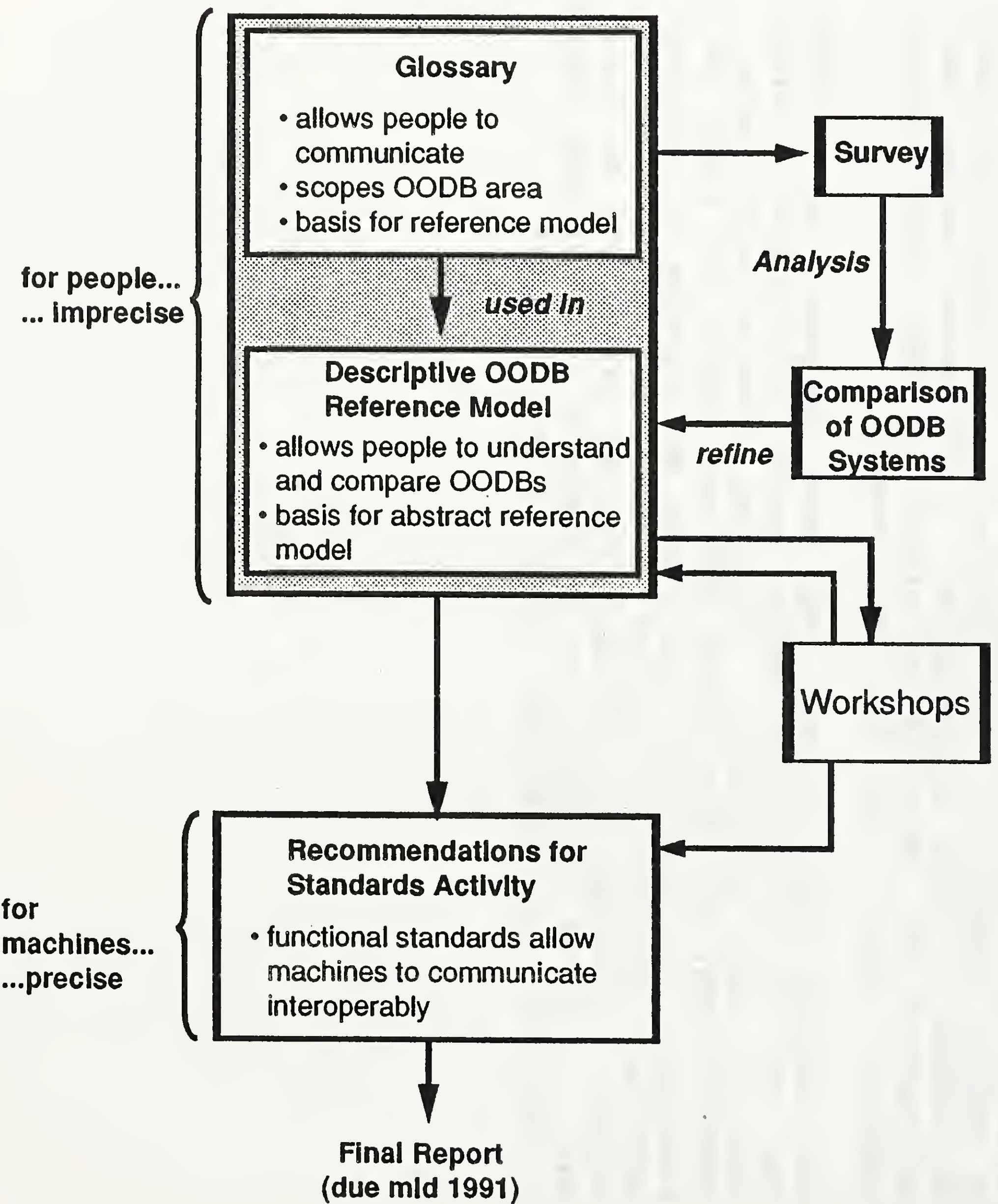
We are seeking detailed position papers on OODB topics including the following

- Object Data Model
- Persistent Language
- Query and Data Manipulation Languages
- Persistent Object Store
- OODB System Architecture and interfaces
- Distributed Object Transport Protocol
- Change Management
- Transactions

Papers should be specific, focusing on concrete proposals for language or module interfaces, exchange mechanisms, abstract specifications, common libraries, or benchmarks. They should identify capabilities that it may be possible to reach community consensus on. Papers on relationships of these OODB capabilities to existing standards are also welcome as are papers that question the wisdom of OODB standardization. OODB system designers and implementers are especially encouraged to share their practical experience where it can lead to consensus.

The workshop itself will begin with some background on OODBTG, then shift to presentations and/or panels with speakers invited based on paper submissions. Workshop attendees will receive OODBTG draft documents (OODB Reference Model, etc) before the workshop. All contributed papers will be available to workshop participants and will be identified in the OODBTG document register. Following the workshop, selected papers will be bound into a workshop proceedings to be available as a National Institute of Standards and Technology (NIST) technical report.

X3/SPARC/DBSSG OODB Task Group Workshop



X3/SPARC/DBSSG/OODBTG OBJECTIVES

- Define a common reference model for OODBs
- Assess where standardization on OODBs is possible and useful
- Complete a Final Report in 1991 containing recommendations regarding future ASC/X3 standards activities in the OODB area, including how these standards would relate to existing standards.

OODBTG meets quarterly. Contact Elizabeth Fong, National Institute of Standards and Technology (NIST), Tech. Bldg. A266, Gaithersburg, MD 20899 (301-975-3250)

MAY 22 WORKSHOP OBJECTIVES

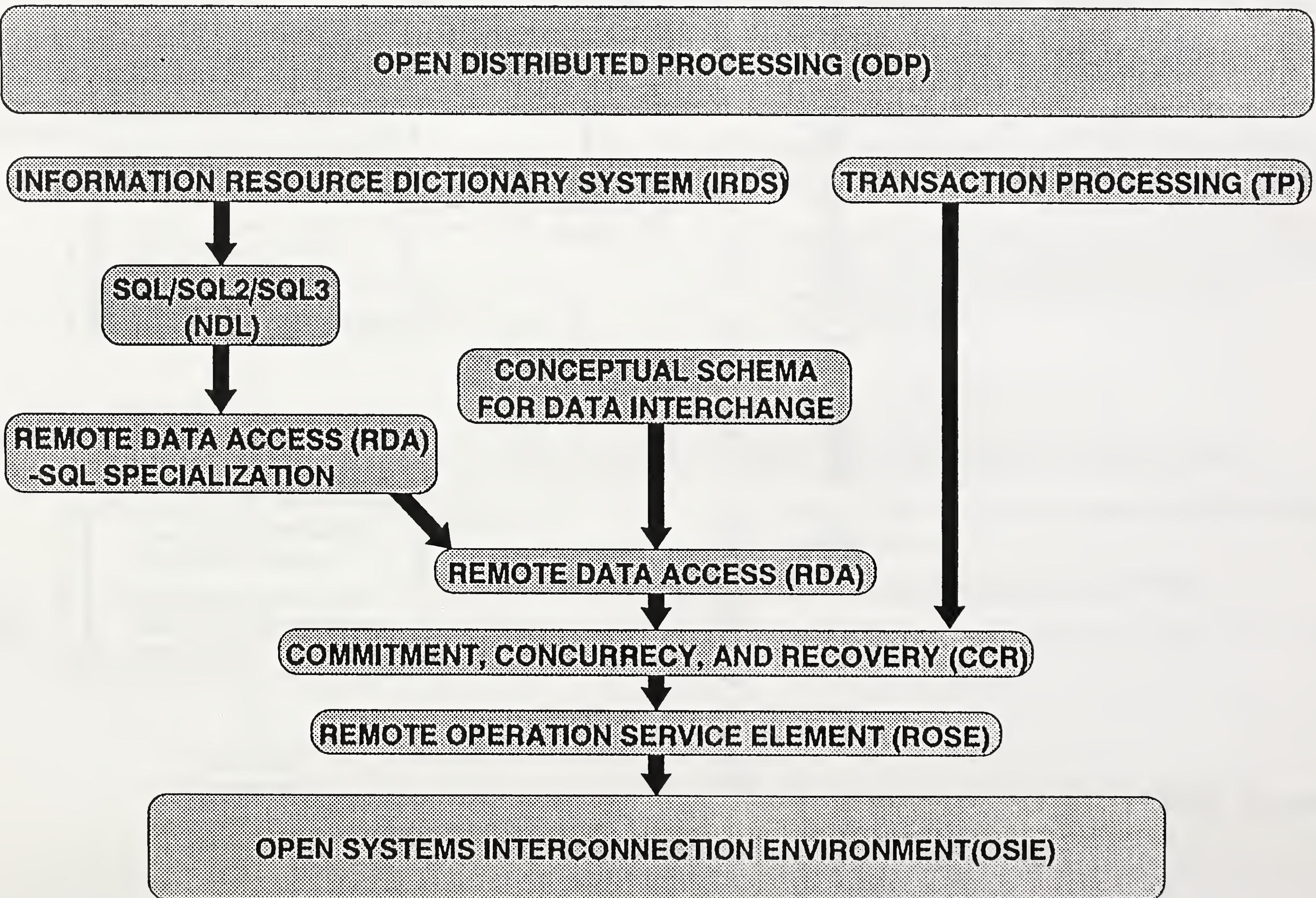
- Review OODBTG Draft Reference Model and get public feedback
- Identify specific standards work items
- Develop liasons with OMG, CFI, X3H2 SQL, X3J16 C++, etc.

OCT. 23 WORKSHOP IN OTTAWA PLANNED

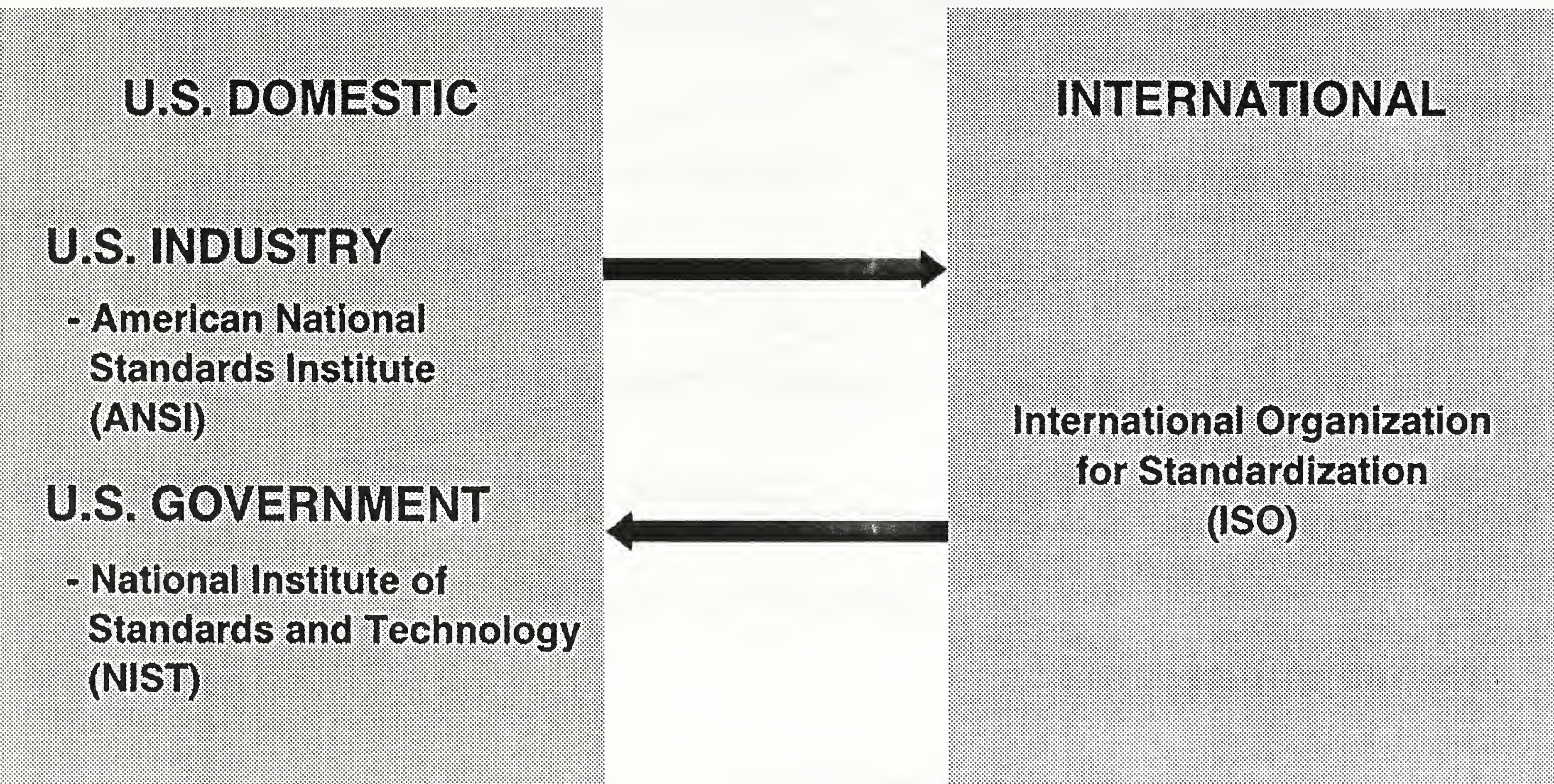
The Role of Standards

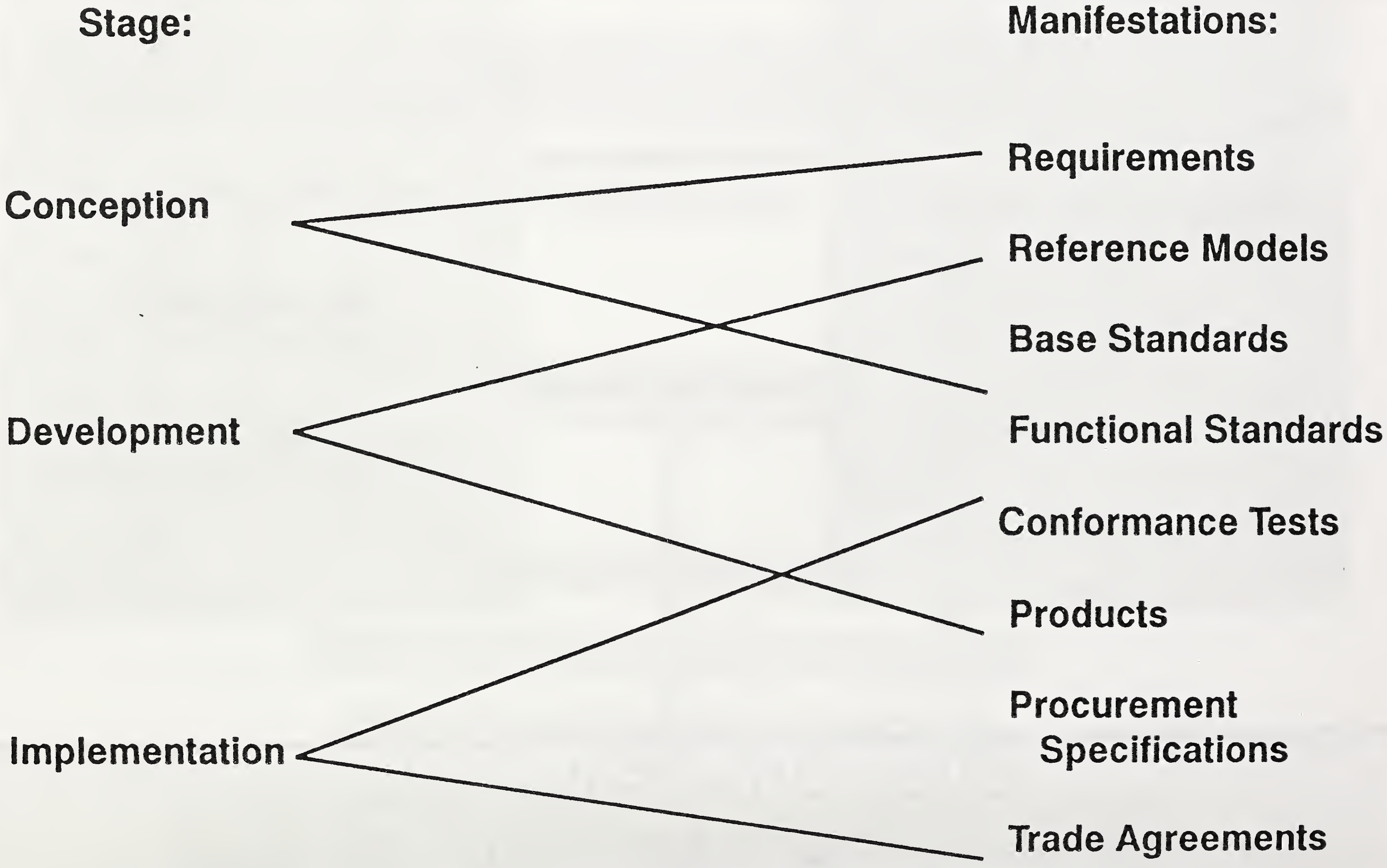
X3/SPARC/DBSSG Chairman Ed Stull, Summa Intl.

EVOLVING DATABASE-RELATED STANDARDS



U.S. DOMESTIC STANDARDS PROCESSING



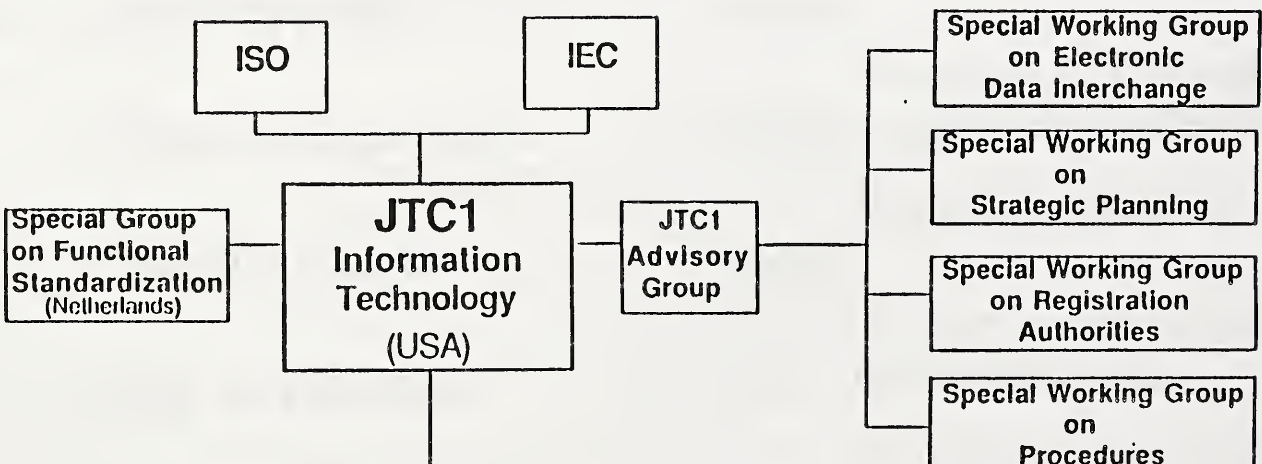

Life Cycle of IT Voluntary Standards
Stage:
Manifestations:
Conception
Development
Implementation
Requirements
Reference Models
Base Standards
Functional Standards
Conformance Tests
Products
Procurement Specifications
Trade Agreements

International IT Voluntary Standards Organizations

Arena	*Type of Member*	*U.S. Representative*
ISO	member body	ANSI
IEC	national committee	USNC (ANSI Secretariat)
ISO/IEC JTC1	national body	ANSI
CCITT	national government	State Department

June 1989

ORGANIZATION CHART FOR ISO/IEC JOINT TECHNICAL COMMITTEE 1 (JTC1) INFORMATION TECHNOLOGY



Application Elements Group

SC1	SC7	SC14	SC22
Vocabulary	Design & Documentation of Computer Based Info. Systems	Representation of Data Elements	Languages
(France)	(Canada)	(Sweden)	(Canada)

Equipment & Media Group

SC11	SC17	SC23	SC28
Flexible Magnetic Media for Digital Data Interchange	Identification and Credit Cards	Optical Digital Data Disks	Office Equipment
(USA)	(United Kingdom)	(Japan)	(Switzerland)

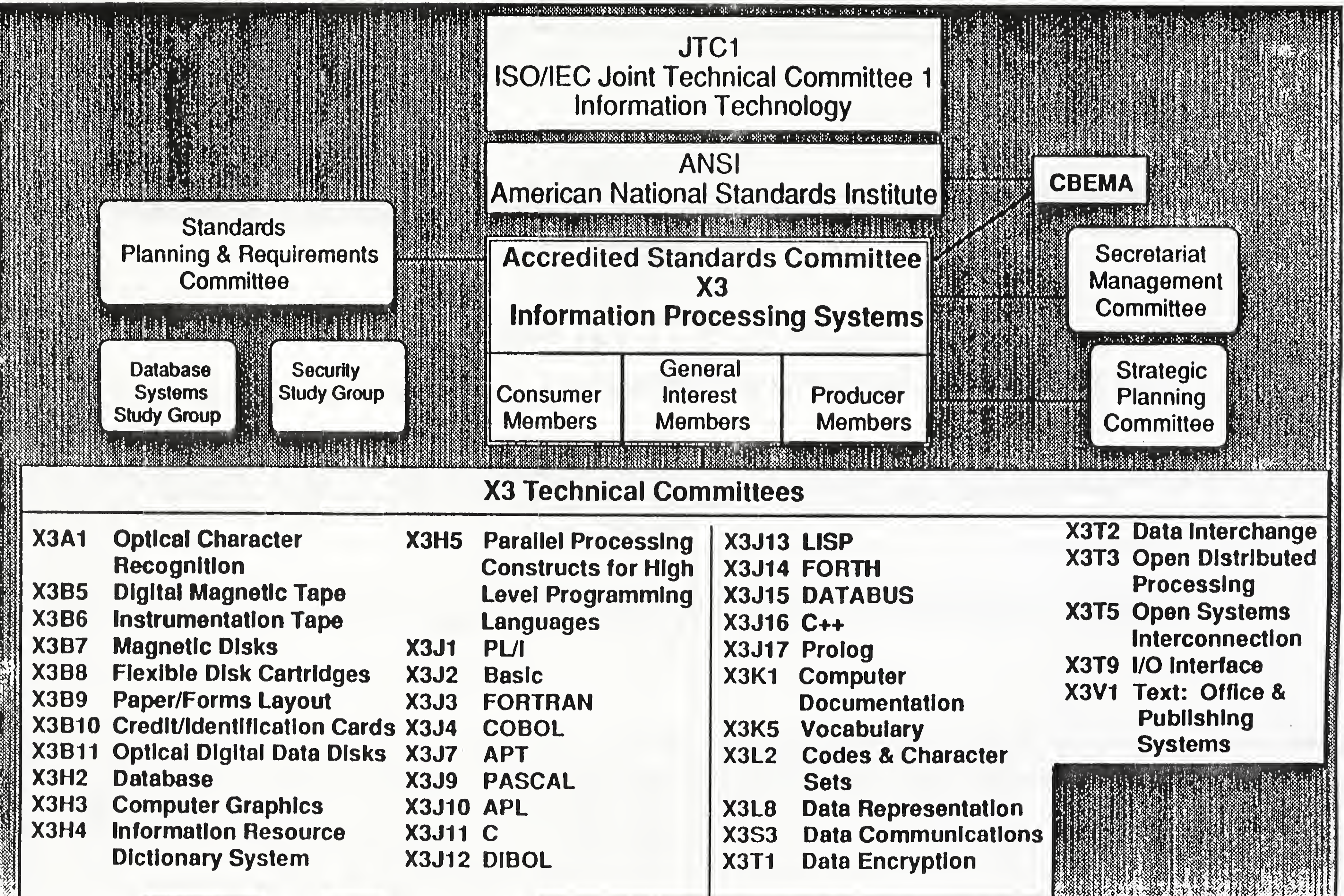
Systems Group

SC6	SC18	SC21	SC26
Telecommunications and Info. Exchange Between Systems	Text & Office Systems	Information Retrieval, Transfer and Management for OSI	Microprocessor Systems
(USA)	(USA)	(USA)	(Japan)

Systems Support Group

SC2	SC24	SC25	SC27
Character Sets and Information Coding	Computer Graphics	(Merger of old SC13 & SC83)	IT Security Techniques
(France)	(Germany, F.R)	(Germany, F.R)	(Germany, F.R)

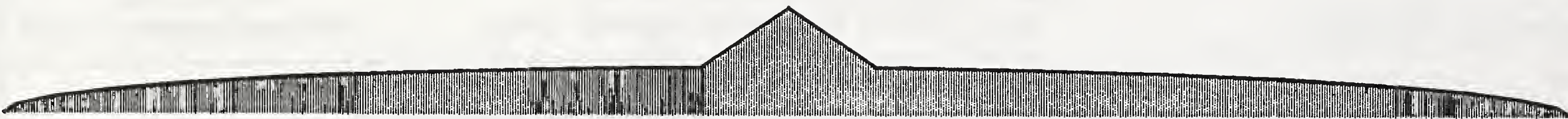
Organizational Chart for X3



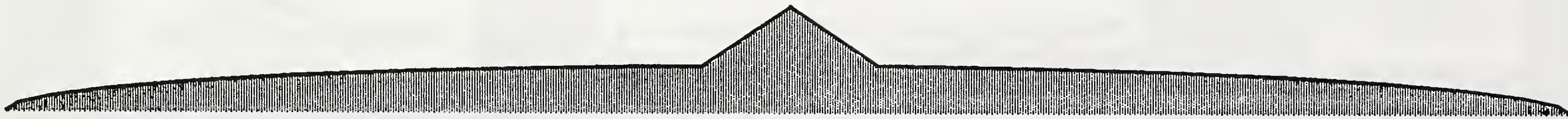
Summa International, Inc.

INTERNATIONAL PROCESSING FOR DATABASE-RELATED STANDARDS

International Organization for Standardization (ISO)



U.S. SC21 Combined Technical Advisory Group (TAG) develops the U.S. Position



X3H2
Data Languages

DBSSG
Data Management Standards Planning

X3H4
Information Resource Dictionary Systems

ORGANIZATIONS INVOLVED IN THE DBSSG

DBSSG and Task Group Total Members: 108 +

1st American Bankshares
Alberta Workers Comp. Board
Allied Signal Corporation
Andersen Consulting
Argonne National Laboratory

AT&T
AT&T Bell Labs
BBN
Bell Communications Research
Booz-Allen & Hamilton

C2 Systems, USAF/RADC
Chase Manhattan Bank
CIS IUPUI
Citicorp International Comm.
Computer Associates, Intl.

Computer Sciences Corporation
Computer Technology Planning
Consultant
Coopers & Lybrand
Datawise Inc.

DBC
Dept CB, Cal. State University
DoD Computer Institute
DOD
DOE Computer Security, Los Alamos

DOT/FAA
East Texas State Univ. & MDC
EDP News Service
Ernst & Whinney
Fed. Emergency Mgmt. Agency

Federal Emergency Mgmt
Federal Reserve Bank of Dallas
GTE
Harris Corporation
Hewlett Packard Laboratories

Honeywell Inc.
Hughes Aircraft Company
IBM Watson Research Laboratory
IEEE
Information Impact Intl, Inc.

INTERCON Systems Corporation
JPL
Kanna Associates
Natl. Computer Security Center
MITRE Corporation

Natl. Computer Security Center
Natl. Computer Security Center
NIST NCSL
Objectivity Inc.
Ontologic, Inc.

ORACLE Corporation Canada
P.H. Hagey Mgmt Consulting Ltd.
Pacific Bell
RAND Corporation
RC Smithsonian

Relational Technology, Inc.
Robert Courtney, Inc.
SAIC
Sandia National Laboratory
Secure Systems, Inc.

Servio Logic Development Corp.
SIAM
Skyline Information Systems
Softbridge Microsystems Corp.
SRI International

Summa 8 (USA)
Texas Instruments
Thomsen Due Associates, Ltd
Treasury Department
TRW, Inc.

U.S. West
U.S. West Advance Technologies
UNISYS
UNISYS General Systems Group
University of Minnesota

DBSSG TASK GROUPS

Glossary	GTG	standardizing the terminology of databases
Common Data Model	CDMTG	developing a unified set of structures and functions of which other data models and DBMSs are subsets
Future Advances in Database Systems	FADTG	investigating and determining the need for standards on advanced concepts
Data Analysis and Design	DADTG	studying data analysis and design methodologies, techniques, and tools
Data Management Security and Privacy	DMSPTG	investigating DoD, federal, and commercial needs for security and privacy in database systems
Object Oriented Database	OODBTG	investigating the technology of object-oriented database management systems
Open Systems Interrconnection and Database	OSIDBTG	investigating the integration of OSI and database standards

OODBTG Charter, Progress, and Plan

OODBTG Chairman Tim Andrews, Ontologic

PLAN AND ORGANIZATION FOR OBJECT-ORIENTED DATABASE TASK GROUP (OODBTG)

OBJECTIVE:

In recognition that the rapid emergence of a new information management technology, called Object Database Systems, and the lack of a common definition of these terms is causing confusion among the vendors, users and standards developers in the database community, the Object-Oriented Database Task Group seeks:

1) To establish a working definition for the term "Object Database,"

2) To establish the relationship between Object Database technology and "object-oriented" methods and technology in other fields, including programming languages, user interface methodologies and information modeling methodologies, and

3) To establish a framework for future data management standards activities, both extensions to ongoing SQL and IRDS development, and related future standards.

These results are to be presented in the form of a Technical Report on Object Database Technology and a Recommendation For Standards Planning.

SCOPE:

The Task Group will review existing and under development products claiming to be object databases or object-oriented products with database capabilities. The Task Group will also review published literature and research activities concerning object database technology world wide, and will seek to generate discussion among practitioners in the field.

OPERATIONAL APPROACH:

The Task Group divided the work into five related areas:

- o Glossary definition of object-oriented terms.
- o Compilation of object-oriented bibliographic references.
- o Identification of characteristics and features for object database managmenet systems.

- o Survey of commercial and research prototype object database management systems using the features identified.

- o Reference model for object database management system.

- o Issues relating to recommendations to data management standards planning.

The Task Group plans to meet its objectives in about two years. The final product will be the publication of the Technical Report and recommendations for standards planning.

The Task Group plans to hold quarterly meetings, in conjunction with DBSSG quarterly meetings when possible. Additional geographic subgroup meetings will be held to permit more technical work progression. Subgroups may be created on an ad-hoc basis by the Task Group Chairperson. The Task Group Chairperson will report quarterly, either directly or through a representative, to the DBSSG, and will include in the report a list of active subgroups and the issues being addressed, and a statement of progress made during the previous quarter.

Workshops and panels are also planned to generate discussion among practitioners in the database and programming language fields.

PLANNED MILESTONES AND SCHEDULES:

Jan 1989: Formation of the Task Group by DBSSG

Apr 1989: Statement of Issues to be Addressed. Listing and classification of Concepts going by the name Object Database

Jul 1989: Outline for Reference Model and Technical Report

Jan 1990: Initial Draft Tutorial Collected.
Initial features explained for the survey form.

Apr 1990: Initial Draft Reference Model Collected.
Preliminary data collected for a few OODBMSs

Jul 1990: Draft Reference Model and Recommendations submitted to DBSSG

Oct 1990: Comments and Requests for Revision back to the Task Group

Jan 1991: Final Technical Report and Recommendations Submitted to DBSSG for submission to X3/SPARC

Jul 1991: ODBTG Disbanded after responding to questions and comments from SPARC and X3 Technical Committees

Object Database Task Group Organization

Chairperson: Timothy Andrews
Ontologic Inc
47 Manning Road
Billerica, MA 01821
(508) 667-2383

Vice Chairperson: Bill Kent
Hewlett Packard Laboratories
1501 Page Mill Road
P.O. Box 10490
Palo Alto, CA 94303-0969
(415) 857-8723

Secretary: (designated at each meeting)

Correspondence Secretary: Elizabeth Fong
National Inst. of Standards & Technology
Technology Building A266
Gaithersburg, MD 20899
(301) 975-3250

Reference Model Editor: Allen Otis
Servio Logic Development Corp.

OODBMS Survey Subgroup Liaison: Gordon Everest
University of Minnesota

Bibliography Subgroup Liaison: Mark Sastry
Honeywell Inc.

Glossary Subgroup Liaison: Roy Gates
Rand Corporation

MEMBERSHIP:

As specified in X3 Procedures, members of OODBTG serve as individual experts. The rules for OODBTG membership is the same as memebrship for DBSSG.

The OODBTG has divided its membership into "active" and "correspondence" members. The active members need to attend at least two consecutive meetings in order to be qualified as "active." "Correspondence" members are those who are interested in the activities of OODBTG, but may not wish to attend and participate in meetings.

OODBTG Reference Model (Draft)

Editor: Allen Otis, Servio Corporation

Accredited Standards Committee Document Number: OODB 89-01R4
X3, INFORMATION PROCESSING SYSTEMS

Project: DBSSG/OODBTG
Ref Doc: "ODM Reference Model", OODB 89-01R3,
Reply to:
Allen Otis,
Servio Corporation
15220 NW Greenbrier #100
Beaverton, OR 97006

503 629 8383 / FAX 503 629 8556
otisa@servio.slc.com
uunet!servio!otisa

Date: 6 May, 1990

TO: OODBTG members

FROM: Allen Otis

SUBJECT: Revised draft of Reference Model for Object Data Management

This document is a working draft that will become a section of the final report of the Object Oriented Database Task Group (OODBTG) of the Database Systems Study Group (DBSSG).

This revision of the reference model is based on discussions at the April OODBTG Meeting in Berkeley, and will be distributed to attendees of the workshop in May.

A Reference Model for Object Data Management

Contents

Preface 3

1. Introduction 5
   - 1.1 Motivation 5
   - 1.2 Purpose of Document 5
   - 1.3 Intended Audience 6
   - 1.4 Relationship to Standards Activities 6
   - 1.5 Organization of the document 7

2. The Object Data Management Paradigm 8
   - 2.1 Overview 8
   - 2.2 Object Paradigm Characteristics 9
     - 2.2.1 Encapsulation 10
     - 2.2.2 Objects, Behavior, Messages, and States 10
     - 2.2.3 Identity 12
     - 2.2.4 Types and Classes 14
     - 2.2.5 Composition 14
     - 2.2.6 Binding and Polymorphism 15
     - 2.2.7 Inheritance and Delegation 16
     - 2.2.9 Extensibility 17
     - 2.2.10. Information semantics 17
   - 2.3 Database Paradigm Characteristics 18
     - 2.3.1 Persistence and Object Lifetimes 18
     - 2.3.2 Data Language 20
     - 2.3.3 Integrity Constraints 21
     - 2.3.4 Transactions and Concurrency Control 22
     - 2.3.5 Recovery 22
     - 2.3.6 Versions and Change Management 23
     - 2.3.7 Distribution 23
     - 2.3.8 Security 24

3. ODM System Interfaces 24
   - 3.1 User Roles and Class Libraries 24
   - 3.2 Application Program Interfaces 27

A Reference Model for Object Data Management

Preface

This is a report produced by the Object-Oriented Databases Task Group (OODBTG) of the Database Systems Study Group (DBSSG). The DBSSG is one of the advisory groups to the Accredited Standards Committee X3 (ASC/X3), Standards Planning and Requirements Committee (SPARC), operating under the procedures of the American National Standards Institute (ANSI).

> **Note:** This document is currently at the working draft stage, and has not been approved by the OODBTG nor by DBSSG for incorporation in the final report of the OODBTG. OODBTG welcomes comments, and invites attendance by technical contributors at our meetings and workshops.
>
> In this report, we have elected to use the term **Object** rather than **Object-Oriented.** We have decided not to spend any of our effort on the formalities required to change the name of the task group, which remains OODBTG.

The OODBTG was established in January 1989. Consistent with usual practice when confronted with a complex subject, DBSSG charged the OODBTG to investigate the subject of Object Databases with the objective of determining which, if any, aspects of such systems are, at present, suitable candidates for the development of standards.

Object Databases represent a rapidly emerging technology which combines object software and database management technologies. Object technology has now reached a point where formal standardization is a viable option. This report represents the consensus of members of OODBTG to initiate some specialized standardization efforts. The purposes of this technical report are:

* To state informal definitions and requirements for an Object Database Management System, which can be used as a base document by a specialized standardization technical committee.
* To define a common reference model for an Object Database, based on object software and database management systems models.
* To enable OODBTG to make recommendations regarding future ASC/X3 standards activities in the areas of data management, data languages such as SQL, IRDS, ODP development, and related standards.

This technical report represents the work of many individuals who attended quarterly meetings in conjunction with DBSSG meetings, and several ad-hoc subgroup meetings held in the east coast, west coast, and midwestern regions. The technical work represents the careful distillation of direct contributions by the members of OODBTG. The opinions and ideas expressed here are not necessarily endorsed by all of the members nor by the members' sponsoring organizations.

This report is being edited by:
Allen Otis, Servio Corp.

In addition, regional subgroups of OODBTG held technical meetings to develop the document. These meetings were chaired by:
Tim Andrews, Ontologic
Gordon Everest, University of Minnesota
William Kent, Hewlett Packard

The following individuals have attended at least one of the task group meetings and have made technical contributions to the report:
Stephanie Cammarata, Rand Corporation
Richard T. Due, Thomsen Due & Associates Ltd.
Larry English, Information Impact International Inc.
Elizabeth Fong, NIST
Roy Gates, Rand Corporation
Pat Hagey, P.H. Hagey Consulting Ltd.
Sandra Heiler, Xerox
Ken Jacobs, ORACLE
Michael Jende, UNISYS
Haim Kilov, Bellcore
William McKenney, Workers Compensation Board
Ken Moore, Digital Equipment Corporation
Bhadra Patel, Hughes Aircraft
Girish Pathak, Xerox
Paul Perkovic, Informix Software Inc.
Gary Rivord, Sandia National Labs
Katie Rotzell, Object Sciences
Mark Sastry, Honeywell, Inc.
Jay Smith, Anderson Consulting
Tom Soon, Pacific Bell
Ed Stull, Summa II
Craig Thompson, Texas Instruments
Andrew Wade, Objectivity Inc.
Miya Yuen, Andersen Consulting

1. Introduction

This section summarizes the motivation behind the development of Object Data Management technology. It also summarizes the purpose, intended audience, related activities, and organization of this document.

1.1 Motivation

There are several threads in the roots and motivation of object database.

One thread is the need to improve various phases of application management, including development, maintenance, enhancement, portability, information integrity, extensibility, and interoperability. Another is the emergence of new classes of applications in such areas as office automation, document processing, and various engineering disciplines. These new applications involve managing complex objects, behavior of objects, and multimedia forms of information. These areas impose new requirements in terms of the complexity in the nature and form of information to be represented and manipulated, as well as more elaborate modes of data sharing.

In programming, the use of objects emerged as a technique that made more complex data types available to programs while simultaneously making programs less aware of, and less dependent on, details of how such structures are implemented. More recently recognized is a need for such objects to be made available to other programs than the ones which created them, i.e., for persistent objects. The first approach to persistence is to store such objects in files. Soon, however, the need arose for additional functionality, such as recovery, concurrency control, access control, and query -- i.e., database.

Thus one aspect of object database is an improved approach to the development of applications and the management of information, involving such principles as abstraction, identity, classification, inheritance, and so on. Object database has also come to include other responses to the application environments, such as change management (versions and configurations), different transaction concepts in support of long and group work.

1.2 Purpose of Document

This document defines a reference model for Object Data Management Systems (ODM systems) and provides a framework within which we can distinguish an ODM system from other data management systems.

This document is an abstract description that will support concrete specifications of multiple ODM systems. This document forms a reference model from which to recommend future standards activities.

Good examples of successful reference models, not necessarily related to Object Databases are the OSI 7-layer reference model for data communications, and the ANSI/SPARC 3-schema architecture.

1.3 Intended Audience

The intended audience for this reference model consists of:

1. Those who are managing the standards process and will influence the direction of further work in the area of object databases.

2. Those who are developing standards based on this reference model.

3. Those who are building or using products relevant to standards proposed by this reference model.

4. Those seeking to understand what the object paradigm adds to the traditional database paradigm and data management systems

1.4 Relationship to Standards Activities

It is necessary to put this Reference Model document into the context of the overall OODBTG charter and statement of work. The Final Technical Report of the OODBTG, due to its parent organization ASC X3/SPARC/DBSSG in mid 1991, must recommend what standards would be reasonable and useful in the area of ODM. This Reference Model document is an important instrument in gaining consensus to answer this question. It provides a basis and framework for comparing different ODM systems.

Other activities related to OODBTG are:

o A Survey of ODM systems. The data collected by OODBTG in this survey will be analyzed to provide refinements to the Reference Model including dimensions of comparison for how ODM systems differ. .

o Workshops on the potential for ODM Standardization which will provide data points for consideration as possible ODM standards. As such, they may provide one or more concrete extensions of the abstract reference model. If enough consensus exists that these proposed standards will be useful, they will be included in the recommendations of the OODBTG Final Report. OODBTG is currently planning to hold workshops in May 1990 and October 1990.

o The 3rd Joint Standards Meeting at Anaheim, CA, January 1991. The topic of this meeting is **Objects in Data Management**. This meeting brings together many different technical committees working on standards related to data management. A goal of this meeting is to identify and possibly resolve conflicting issues with respect to the object paradigm. DBSSG is one of the organizers of this meeting.

1.5 Organization of the document

Following the introduction, this document consists of two major sections, the Object Data Management Paradigm, and User Roles and Interfaces. The ODM Paradigm section explains the fundamental concepts embodied of this paradigm. The User Roles and Interfaces section discusses interfaces and users' roles envisioned for ODM systems.

Recommendations for the future standards activities will appear in the OODBTG Technical Report, which is the parent document to this document.

2. The Object Data Management Paradigm

The section contains Overview, Object Paradigm, and Database Paradigm subsections, and explains the fundamental concepts embodied in the ODM Paradigm.

2.1 Overview

As described in Section 1, a **reference model** provides a framework that **defines** a system, process, or other artifact, providing criteria and features that **cover** important aspects of existing and future systems, thus permitting them to be compared. In addition, a reference model must be useful in providing a road map for identifying and developing concrete standards.

This section presents a reference model for ODM systems. The framework is presented as a **design space** of characteristics (features, capabilities, functions).

A design space is a methodological tool that provides an organized way to record the space of design characteristics and design decisions. It takes the form of an AND-OR graph. AND layers show how a module is composed into functions, parts, or characteristics. OR layers represent alternative ways to realize a function. AND layers show similarities shared by all decompositions. OR layers provide dimensions of comparison. The upper layers of a design space are more abstract. As the space is refined, it covers more system-specific and concrete design details.

Figure 1 shows the design space for ODM systems. Only the top AND-layer is shown. The remainder of section 2 describes these characteristics in detail. As can be seen in the diagram, the characteristics are grouped by heritage, since ODM systems inherit both object and database characteristics.

Based on the consensus of the OODBTG, the characteristics shown in Figure 1 are the commonly shared characteristics of ODM systems. **The design space we describe is maximal in the sense that not all ODM systems have all features in this AND layer.** For instance, some ODM systems may not have user interfaces. In addition, some of the characteristics (e.g. transactions, recovery, distribution, security) appear to be largely data model independent; that is, they do not differ from similar notions for other relational and other databases, except as noted in the related sections. Finally, OR layers are not shown in Figure 1. They provide differentia, or dimensions of comparison, where different ODM systems may take different approaches in defining a characteristic. Some ODM systems may implement more than one OR-alternative (e.g. both pessimistic and optimistic transactions). Since only the top level of the design space is described in this document, it is most accurate to consider that our reference model to date is an "abstract reference model" for ODM systems. That is, at this level of detail, it provides a way to compare ODM systems but does not provide an implementable specification.

Object Data Management

Object Paradigm Characteristics:
|---Encapsulation
|---Objects
|---Identity
|---Types and Classes
|---Composition
|---Polymorphism
|---Inheritance
|---Extensibility

Data Management Paradigm Characteristics:
|---Persistence
|---Data Language
|---Integrity Constraints
|---Transactions
|---Recovery
|---Versions
|---Distribution
|---Security

User Interfaces

Figure 1: ODM Design Space

It is worth noting that the ODM design space includes elements from both the Programming Language and Database sectors of computer science. One of the goals of this document is to describe the ODM design space in a way that accommodates both the programming language and database perspectives of ODM systems.

The rest of section 2 describes the OODB Reference Model and is structured to match Figure 1. A consequence is that Section 2.2 on the object data model or subsections of Section 2.3 on the database paradigm could almost stand alone as mini-reference models. For example, the description of objects in Section 2.2 could be used in other reference models having little to do with databases, such as interoperability frameworks, object-oriented programming languages, or remote data access facilities. The subsections of 2.3 could, when developed in more detail, find independent use in reference models dealing with such topics as persistence, transactions, query languages, or change management.

2.2 Object Paradigm Characteristics

In the generalized Object Paradigm we define the following terms:

> Encapsulation, Objects, Identity, Types and Classes, Composition, Polymorphism, Inheritance, and Extensibility

These terms are defined independently of any distinction between Programming and Database domains.

2.2.1 Encapsulation

The essence of the object paradigm is encapsulation, i.e. a clear separation between the external behavioral semantics of objects and their internal implementation. Encapsulation means that a black box specification of objects is provided, and that the internal implementation, state variables, or data are not visible in this specification. In this spirit, the very nature of objects can be defined behaviorally, in terms of the behavior of executing operations. The generalized behavior, when progressively restricted, corresponds to various forms of such object concepts as messaging and state. The approach is general enough to encompass notions of objects both as static (passive) recipients of messages and as dynamic (active) agents capable of such activities as sending messages.

The core concept of the object paradigm may be explained in terms of operations. An object is something which can play a role in an operation, either as an operand, as its result, or as the method which performs the operation. For example:

> An operation is applied to an object, which may be a simple object or an aggregate of other objects.
>
> Application of the operation causes a method to be executed.
>
> The effect of executing the method may be to return a result object (simple or aggregate) and/or to modify the result the some other operation will produce.

The users of objects may be programs or people. Such users observe the behavior of objects in terms of the operations which may be applied to objects, and in terms of the result objects returned by such operations. An operation may be implemented (i.e. supported or realized) by a variety of different program code and data structures. Encapsulation means that these implementations are not visible to the user of the object.

We describe these concepts in more detail in the following sections

2.2.2 Objects, Behavior, Messages, and States

Most object models define the concept of "object" in a way which corresponds to one or more of the definitions in the following list of increasingly restrictive definitions. A "thing" is an object if it fits one or more of these definitions.

The most general concept of an object is the **generalized operation model** of an object:

1. An object is something which plays a role in an operation. The object may be an operand, a result, or a performer of an operation. Operations may return large or small literal values (numbers, strings), aggregate objects (sets, lists), or any other kinds of objects.

To define objects only in terms of their visible external behavior, we exclude performer objects:

2. An object is an operand or result of an operation.

In (2), an object is accessible only through its **behavior**, which consists of all operations defined for the object. The list of defined operations is not necessarily fixed, and is often changeable at run-time. However only an operation may not be applied to an object before it has been defined.

Focusing on objects as the passive subjects of activity, we arrive at this restricted notion of object:

3. An object is an operand of an operation. Thus, for example, in a Connect(wire1,pin2) operation, wire1 is the first operand and pin2 is the second operand.

   As another example consider document, person and chapter objects. The operation AssignAuthor(document, person) changes the author of the document. The operation Author(document) returns the person who is the author of the document. AddChapter(document, chapter) adds a chapter to the document. Length(document) returns the number of chapters in the document.

The **messaging model** is introduced by distinguishing one of the operands as the recipient of the operation. The operation can then be called a message, and the other operands are then called parameters. In the messaging model, **behavior** is defined in terms of how the object responds to messages.

4. An object is something which is a distinguished operand of an operation, i.e. the recipient of a message.

A common syntactic convention for (4) is to so distinguish the first operand. Under this convention, the Connect(wire1,pin2) operation is considered a message to wire1, taking pin2 as a parameter. Another common syntax places the recipient first: wire1.Connect(pin2). Or, AddChapter(document, chapter) has document as the receiver and AddChapter as the message.

The operation and messaging paradigms are indistinguishable for unary (single-operand) operations. Thus Length(wire1) is both an operation applied to wire1 and a message sent to wire1, though in message syntax it may take the form wire1.Length.

The result objects returned by some operations may change over time. Object **state** may be characterized by the result of such **accessing operations** at a particular time. Change of state may be described in terms **state-changing operations** which alter the subsequent results of accessing operations. Object state may be implemented by various configurations of stored data structures inside of objects for recording the effects of state-changing operations. For example:

> If the accessing operation IsConnected(wire1,pin2) returns True or False depending on whether they are connected, then the result of IsConnected may be changed by the state-changing operation Connect(wire1,pin2). AddChapter(document, chapter) is a state-changing operation and Length(document) is an accessing operation.

This leads to a **behavioral characterization of state** which is a refinement of (3):

5. Objects are things whose states are reflected in the results of accessing operations. These objects have state-changing operations which will change the subsequent results of at least some of the accessing operations.

In (5), we have one kind of encapsulation. The states of objects are known only by the results of operations, and not in terms of internal structure. The states of objects are thus collectively encapsulated from users of the objects but not necessarily from other closely related objects. State may jointly characterize several objects, such as wires and pins from the connection example.

A stronger form of encapsulation, common in many object models, also isolates the states of objects from each other. The operation IsConnected(wire1,pin2), perceived as a message to wire1, is perceived to reflect the state of wire1 but not of pin2. This isolated-state encapsulation model is the intersection of (4) and (5):

6. Objects are things whose states are reflected in the current values of accessing messages for which the objects are receivers.

In (6) the implementation of objects is disjoint between objects; all methods and states belong exclusively to individual objects.

If all of the accessing operations have only one operand (i.e. are unary), such as Length, then (5) and (6) are the same.

Some objects may have accessing operations but no state-changing operations, either because the objects contain no state, or because the objects are immutable. For example, a literal number such as the integer 25, has no state-changing operations, but might have an accessing operation such as Print that returns an ASCII string representation of the integer.

2.2.3 Identity

Identity is a semantic concept. Identity provides a means to denote an object independent of its behavior, state, or content. The concept of identity is independent of which definition of "Object" is being used. The definition of identity includes the concepts of identity compare operation, logical identifiers, object identifiers, object creation, literal values, and aggregates.

Let X and Y refer to objects. The identity concept can be described in terms of an identity compare operation == such that X == Y implies

A) h(X) and h(Y) have the same result for any operation h(), where h() has no side effects (i.e. h() is not a state-changing method of either X or Y), and

B) the set { X, Y } has a cardinality of 1, i.e. it is really { X }

NOTE: In this section we are using the mathematical definition of a set taken from set theory, rather than any of the computer science or computer language related definitions of sets.

A common realization of object references is by means of logical identifiers. We use the term **logical identifier** (LID) to mean an implementation-free token that is uniquely associated with a specific object. The term **object identifier** (OID) frequently means a specific implementation of a logical identifier using a fixed number of bits.

A LID is meaningful within some limited scope which we shall call an object space. A LID from one object space may not be a valid LID in another object space, or may identify a different object. Therefore, identity comparison between LIDs must take into consideration whether or not they are in the same space.

An object identifier is unique within a particular object space.

Given A and B are different LIDs (perhaps from different spaces) and given A == B, then if f(A) and f(B) have the same behavior, the operation f is an **object-based** operation, otherwise f is a **value-based** operation. Value based operations are sensitive to the representation of operands; object based operations give the same behavior regardless of representation.

Object creation occurs within an object space. Each creation event must result in a LID that identifies the created object and that does not identify any other object currently existing in that space. The creation event creates a representation of the object in the space.

Literal values are objects such as numbers and character strings. The definitions of what distinguishes a literal from a created object may be implementation dependent.

Literal values are similar to created objects in that both may occur as operands of operations.

Literal values are different from created objects in that:

1. Literals have immutable state.
2. Literals do not have a create operation because their representations are explicitly recognized.

For example, multiple occurrences of the same literal number are all references to the same point on the number line. Operations which return literal values are constructing references to objects, not creating objects.

With respect to identity, there are two kinds of aggregate objects such as sets. **Intentional** sets have an identity based upon their creation event. Two intentional sets may have different identities but the same members. **Extensional** sets have an identity based upon their membership, that is two sets with the same members have the same identity.

If X and Y are the LIDs of extensional sets, then X == Y if and only if X and Y have the same members. For an Intentional set, the membership at any point in time corresponds to an extensional set. Two Intentional sets remain distinct objects if they have the same members. If the membership of an Intentional set changes, the identity of the set does not change. For Intentional sets "shallow equality" is the same as identity, while "deep equality" means having the same members (i.e. having identical extensional sets as membership).

2.2.4 Types and Classes

The object paradigm deals with both abstract external behavior and the implementation of that behavior. Objects may be grouped into types by commonality of behavior, and into classes by commonality of implementations.

The behavior portion of an object definition defines the **protocol** for the object.

In the generalized operation model, the protocol is the set of roles that the object can play in various operations. In the messaging model, the protocol is the set of operations for which this object may be the distinguished operand (message receiver). The protocol does not provide any visibility of the implementation of the behavior. The protocol completely specifies the behavior of an object. Only operations specified in the protocol are allowed; if a data management system allows operations which are not predefined, then the system is violating the principles of abstraction and encapsulation.

A **type** defines the protocol of a similar group of objects; these objects are instances of the type. The type itself may also be an object. The ability to define types is required in order to be able to define semantic integrity constraints for a database schema.

A **class** defines the methods, messages, and properties of a similar group of objects; these objects are instances of the class. A class may be considered an implementation of a type. The class itself may also an object; methods may also be objects. Classes are necessary in an object based system to organize the implementation of a large number of operations, i.e. a large number of methods.

The implementation of the behavior of an object specifies the code, such as procedural code expressed in some programming language, or query which implements the behavior for each specific message. In a class-based system, the code for a message may access private state of the object, may send messages to the object or to other objects, but may not directly access the state of other kinds of objects.

Classes can be defined by the users of an ODM system. A class may be defined in terms of properties, constraints, and behavior of instances, or by intentional inclusion of members (i.e. instances). These new classes appear to be substantially identical to classes defined as part of the basic system, i.e. they can be used in the same manner as any classes provided as part of the basic system by the vendor of the ODM system.

2.2.5 Composition

A **composite object** is an object which is logically composed of other objects, which are its constituent objects. A composite object is a special kind of aggregate object. The constituent objects are often considered to be dependent upon the composite object. **Composition** implies propagation (usually automatic propagation) of operations to the constituent objects, whereas with aggregate objects such as sets, iteration over the members of the set are required to propagate operations to the constituent objects.

An example of composite objects is a parts list and is-part-of relationship. Some parent object is a composite object composed of a list of constituent parts. Each part object has an is-part-of operation which returns the parent object.

There are two approaches to providing the ability to define composite objects, a behavioral approach and a structural approach .

The behavioral approach - Assume that objects are defined by their behavior; i.e. by accessing and state-changing messages. In this case, composition may be provided by defining additional behavior corresponding to each of the new properties being modelled, such as by defining a "listOfSubParts" operation which returns a collection of objects. Operations may be defined which apply transitively to all objects of which an object is composed.

The structural approach - Composite objects may be built by applying constructors to simpler objects. Constructors may be implemented either by defining new classes or by creating new instances. For example, a new class Part may have several properties each of which is an instance of some other class. A Part might have a "listOfSubparts" property, which is a collection of Parts. Constructors are orthogonal to types or classes, any constructor can apply to any class of objects. Thus by creating collections of collections a "composed-of" hierarchy may be created with arbitrary breadth and depth.

The semantics of composite objects may be further characterized as follows:

1. Certain operations return objects which are considered components of a given object. A **Components** operation might return a list of diagrams when applied to a document. A diagram might be a component of several documents. Also, there might be several such operations, corresponding to different "views". For example, one such operation applied to a file cabinet might return some file folders, while another might return the frame, drawers, locks, etc.

2. Certain operations propagate from one object to another. For example, displaying, copying, or destroying a document causes some diagrams to be displayed, copied, or destroyed.

3. Certain properties propagate from one object to another, in a form of inheritance. For example, the authors and typesetting font of a book become the authors and typesetting font of each of the book's chapters.

2.2.6 Binding and Polymorphism

Binding is the act of associating an operation with a method. For example, a message sent to an object must be bound to some method which provides an implementation of that message for that object. The method is then executed in response to the message.

Polymorphism means that binding of an operation involves a choice among multiple implementations of the operation. Polymorphism allows overloading of operations. An operation may be implemented in several alternative methods, typically associated with different classes.

Different systems support different criteria and times for binding an operation to a method for execution. Examples are early binding, such as compile-time binding and late binding, such as run-time binding.

2.2.7 Inheritance and Delegation

Inheritance means new characteristics are derived from existing characteristics .

In **type-type** inheritance, types can be arranged in a graph. Types are nodes in the graph, and unidirectional links between nodes define inheritance paths. Definitions of protocol, behavior, and state may be inherited along paths of the graph. The two most common forms of type-type inheritance are single inheritance, where the graph is a tree, and multiple inheritance, where the graph is a directed acyclic graph. When creating a subtype, properties may be added to those inherited from the supertypes, or they may selectively replace (override or cancel) those from the supertype. Class-class inheritance is similar to type-type inheritance.

> **Note**: A directed acyclic graph (DAG) is a graph where each node has a unidirectional link to one or more nodes, and each node is the target of one or more links from other nodes. It is not possible to follow a series of links from a node and wind up back at the starting node (i.e. there are no cycles). A tree restricts each node to be the target of exactly one link.

In **type-instance** or **class-instance** inheritance, instances of a type or class inherit behavior and an initial state (default or initial values of certain properties) from the type or class.

In type-based or class-based systems, all behavior of an instance is defined by its type or class.

Delegation-based systems, do not differentiate between class and instances. Objects contain definitions and implementations of operations, and selectively delegate certain definitions and/or implementations to other objects. Object to object delegation replaces both type-type inheritance and type-instance inheritance.

Subclasses (subtypes) represent **specialization** of their superclasses (supertypes). Superclasses(supertypes) represent **generalization** of their subclasses (subtypes).

2.2.9 Extensibility

An extensible system provides the ability to define new types, classes, and/or operations. The new types, classes or operations obey the same object semantics as the previously existing ones. There are several kinds of extensibility:

1. New types and classes may be defined.
2. Types and classes may be modified to add new behavior, properties, constraints, or implementations.
3. Existing types and classes may be used as the basis from which to define new ones.
4. Existing objects may be extended by acquiring new types and corresponding classes.

Additional discussion on extensibility is found in section 3.

2.2.10. Information semantics

The encapsulation characteristic makes it possible to define information semantics in an implementation-independent manner to have the semantics enforced by an ODM system rather than by a set of application programs. In this manner, only semantically meaningful operations are applicable to data.

In order to specify operation semantics in a completely implementation-independent manner, it would be necessary to utilize pre-conditions and post-conditions (i.e., Boolean expressions using constants and results of operations). Namely, for each operation the pre-condition specifies the condition which should be TRUE immediately before execution of the operation (otherwise an attempt to execute the operation will result in an error). In the same manner, the post-condition specifies the condition which should be TRUE immediately after execution of the operation, i.e., as a result of its execution.

A pre-condition and a post-condition completely specify an operation. A set of applicable operations completely specifies a type or class.

Most currently existing data management systems and ODM systems specify results of operations either in a natural language or by means of the code used to implement a method. These approaches provide a less rigorous means of specifying behavior than formal pre-conditions and post-conditions approach would allow.

2.3 Database Paradigm Characteristics

This section discusses some of the major characteristics of an ODM system. Some of these characteristics originate with the object paradigm, others have always been part of data management systems but are modified by the object paradigm, and a few are relatively unchanged. The characteristics discussed in the following subsections are essentially orthogonal to each other.

This section does not provide an exhaustive treatment of each characteristic. Rather the goal is to provide a brief description of each characteristic, with emphasis on the aspects which are unique or changed because of the object paradigm.

2.3.1 Persistence and Object Lifetimes

In most programming languages, objects are transient by default. That is they disappear when the program terminates, unless their representations have been explicitly saved in a file. In databases, information is usually persistent by default. Once created, database objects typically persist until they are explicitly deleted.

In the ODM paradigm, new rules about persistence are needed to manage both the transient objects needed for method execution and the persistent desired for database objects. The rules governing persistence and transience of objects are system dependent.

An object is **reachable** if and only if there exists an operation that returns the object as a result. An object is perceived to exist if and only if it is reachable.

The **life** of an object is the period during which it is reachable. An object begins life as the result of a **create** operation. An object's life is ended by applying a **destroy** operation to it. Some objects such as literals, have eternal lives and are neither created nor destroyed. An object which lives longer than the execution of a single transaction or session is usually called a **persistent** object. An object whose life begins and ends within a single transaction or session is usually called a **transient** object.

Destruction of an object makes it unreachable. Operations which render an object unreachable have the effect of destroying the object.

Destruction of an object may occur as a result of an explicit destroy operation for which the object is an operand, or the object may be destroyed implicitly when a destroy operation is applied to some other object

The rules for destruction of objects are system dependent. If existence of object A depends on the existence of objects B and C, then possible rules include:

1. A will be destroyed when both B and C no longer exist.

2. A may be explicitly destroyed while B or C still exist.

3. A may not be explicitly destroyed until both B and C have been destroyed.

If object A depends on the single object D, then possible rules include:

4. An attempt to destroy D results in both D and A being destroyed

5. An attempt to destroy D results in an error (A must be destroyed first)

6. An attempt to explicitly destroy A results in an error unless D is destroyed first

7. Explicit destruction of A is allowed at any time.

The rules for making a transient object persistent, for destroying transient objects, and for destroying persistent objects are system dependent and may include some combination of the above rules. For example, the session may be an object upon which transient objects are dependent. When the session ends, objects whose existence is not dependent upon a persistent object are transient and are destroyed at the end of the session.

Possible rules for making a transient object persistent include:

8. An transient object must be explicitly made persistent by means of a "save" operation

9. At transaction commit, any transient object which is reachable from some persistent object will be automatically made persistent.

At the implementation level, destruction or deletion of objects may involve reclaiming the storage space occupied by such objects and may involve reclaiming for later re-use the LIDs and/or OIDs of such objects.

An ODM system which implements **garbage collection** typically exhibits the following behavior with respect to persistent and/or transient objects:

A) the system has one or more **scavenge** operations which take as an operand one or more root object(s) and returns as a result a list of otherwise unreachable objects. The scavenge operation is a special operation that can find an object not reachable by any other operation.

B) the system automatically (such as when free storage space is low) or under user or program control executes the scavenge operation and makes the storage space occupied by unreachable objects (and possibly their LIDs or OIDs) available for re-use by subsequent create operations.

If the ODM system implements garbage collection and enforces rules 1,3,4, 6, and 9 above, then the ODM system can enforce object identity-based referential integrity. Other combinations of rules may provide referential integrity, but this is one example of a sufficient set of rules for doing so.

2.3.2 Data Language

The Data Language (DL) of an ODM encompasses the Data Definition Language (DDL), Data Manipulation Language (DML), and Query Language facilities of other database paradigms. The DL has the following uses and capabilities:

Schema Definition - The DL is used as the language with which to manipulate the schema. This includes creating, defining, and modifying types, classes, and operations.

Object Manipulation - The DL is used as the language with which to manipulate (i.e. create, query, modify and destroy) instances of types or classes. The DL is also used to invoke other operations on objects. The object manipulation portion of a DL is often computationally complete, in which case, query and general computation expressions may be combined to provide flexible manipulation capabilities.

Method Definition - The DL may be used to specify operations on objects. In the implementation of methods, the object manipulation portion of the DL may be used to manipulate the private data structures of objects. Methods may be implemented in the DL and/or in other programming language(s).

Base Data Language and Class Libraries

In most ODM systems, the base Data Language is very simple and only the most basic schema definition, object manipulation, and method definition constructs are intrinsic to the language. In the limit, only the most basic abilities to define types or classes, define methods, and invoke operations are required elements of the DL. All other capabilities and operations are usually provided by means of extensible class libraries which define operations that provide all higher levels of functionality. The class libraries effectively become part of the schema of the object database.

Query capabilities of the DL

Query is one of the object manipulation operations provided by the DL. Queries are operations that return objects like any other operations. ODM query operations may be extended with respect to traditional query languages in such ways as:

Querying and/or returning complex data structures

Providing a richer, extendible set of query operations

Operating on user defined types and classes of objects.

Making use of encapsulation, inheritance, or operation overloading

In object terms, a sample query operation would have one or more operand objects which are collections of objects, one or more operand objects which represent predicates, and would return a collection of objects as a result.

> **NOTE: Detailed specifications of query languages, be they SQL, object derivatives of SQL, or direct manipulation query languages, are outside the scope of this reference model. The DBSSG/OODBTG Technical Report will make recommendations for future ODM standards activities which may include query languages.**

2.3.3 Integrity Constraints

If Data Language of an ODM system allows definition of consistency rules as part of the schema, and if the ODM enforces these rules with respect to operations that affect objects covered by the consistency rules, then the ODM provides support for **information integrity**.

These consistency rules are known as **integrity constraints**, and are an integral part of the schema of an ODM system. This means that integrity constraints are part of the definition of a type or class. Examples of integrity constraints are:

Behavioral rules

> Only pre-specified operations should be allowed on objects; i.e. the protocol of objects should be enforced.
>
> If protocol is enforced, then the implementation of operations, i.e. the methods of classes will allow expression of arbitrary constraints in terms of behavioral rules.
>
> Constraints expressed in terms of behavioral rules involving multiple operands of operations, or multiple objects.

Typing constraints

> Constraints on the allowed types of operands and results of operations.
>
> Constraints on the allowed types or classes for values of properties of objects.
>
> Uniqueness constraints on values of a property, across the domain of instances of a class, or across members of some collection.

Referential integrity

> If the rules governing Identity, including control over creation of LIDs and OIDs are enforced then referential integrity of identity-based references is maintained.
>
> Enforcement of referential integrity of key-valued references may also apply to ODM systems.

Enforcement of integrity constraints

Enforcement of encapsulation is necessary to ensure integrity in an ODM system.

Constraints may apply to all instances of a class, or only to specific objects.

Depending on the ODM system, constraints may be automatically enforced at run-time, may be enforced at compile time, and/or may be enforced only when a particular message is sent.

2.3.4 Transactions and Concurrency Control

A **transaction** is a sequence of operations which have been grouped into an atomic or indivisible unit of work. **Concurrency control** consists of those mechanisms provided to implement transactions and control the sharing of objects in the presence of multiple simultaneous transactions.

Existing transaction and concurrency control concepts continue to apply in ODM systems. Characteristics of that are unique to or more common in ODM systems include:

Long transactions (lasting hours or days instead of seconds).

Nested transactions.

Cooperative transactions with more flexible rules for sharing objects between transactions.

Optimistic concurrency control in addition to or in place of traditional locking or pessimistic control. In optimistic control, a transaction access objects without locking by assuming that conflict is unlikely and checking for conflict at the end of the transaction, as opposed to assuming that conflict is likely and locking objects at the beginning of or during the transaction.

2.3.5 Recovery

Recovery is the process of producing a database that has consistent contents after failure of a process or failure of underlying computer system hardware or operating system. Recovery includes handling cases where a process was in the middle of modifying the database when the failure occurred.

In ODM systems, recovery means producing a consistent set of persistent objects in the database after a failure.

2.3.6 Versions and Change Management

Versions and Change Management reflect the influence of such application domains as CAD and Document Management. Management of information in these domains usually includes managing and controlling versions of designs or documents. These domains also rely on configuration control, i.e. the grouping of compatible versions of different documents or designs into a higher level document or design.

ODM systems frequently provide operations, types, classes, and/or methods to manage versions of objects. These operations provide means for the application developer or database user to define application-specific operations for configuration control, and propagation of versioning operations. The developers and/or users of these operations must decide on answers to such questions as:

> When creating a new version of a composite object, how deep does the versioning operation propagate into the components of the object?
>
> What is the protocol for referencing a versioned object? Does each version have a separate identity?
>
> How are versions other than the current version referenced?
>
> Do references to a versioned object refer to a specific version, or do they track the "current" version?

2.3.7 Distribution

The distribution characteristic of a data management system is essentially orthogonal to the object paradigm. A **distributed** data management system is one in which information managed by the system or components of the system exist on more than one computer system, and in which the computer systems are connected by communication links. In an ODM system, things which may be distributed include objects, types and/or classes (i.e. the schema), ODM system functions, ODM system processes, and user processes. The granularity of partitioning the distributed things, and the degree of replication are system dependent.

Distribution of objects may involve either making objects the unit of partitioning, such that an object or replica of an object lives on one computer system, or may involve spreading a single object across multiple computer systems.

The extent to which ODM operations are affected by distribution is described by the system-dependent **transparency** attribute of distribution. For example, in non-transparent distribution, a query operation must include location information as one of its operand objects. In transparent distribution, a query operation does not include location information as an operand because the implementation of the query operation is capable of resolving location automatically.

2.3.8 Security

The security characteristic of a data management system is essentially orthogonal to the object paradigm. Security has at least four essential aspects:

1. The subjects or users of the system must be named

2. The subjects must be authenticated, such as through a login procedure at the start of a session or transaction

3. Authorization operations are provided to specify which operations a subject is permitted to perform, and to specify which objects may be operated upon or used as operands of the permitted operations.

4. An auditing operation is needed to record such events as authorization operations, operations completed or denied, and authentication attempts succeeded or failed.

In an ODM system, the security facilities should be integrated with the concepts of encapsulation, objects, inheritance, and identity.

3. ODM System Interfaces

This section discusses the typical interfaces and users' roles which are envisioned for systems within the scope of this reference model. A common characteristic of many ODM systems is that they include one or more user interfaces, in addition to application program interfaces.

The user interfaces may be command-line-oriented, but often are graphical and allow browsing and editing of object types, classes, and/or instances. A variety of generic user interfaces (e.g. forms or reports) and domain or application-specific interfaces will also be common.

3.1 User Roles and Class Libraries

The roles of users of an ODM system may be characterized in terms of:

* User interfaces employed

* Class Libraries visible to the role

* Relationships to other roles

* Placement of the role in a business organization

Under previous database paradigms, users of database systems might play one or more of the following roles. These same roles exist in the ODM paradigm:

End User
Information Modeler or Data Modeler
Application Developer
System Administrator or Database Administrator
System Builder or DBMS Vendor

In addition, the ODM paradigm brings with it some new roles:

Class Library Developer or Vendor
Class Definer
Method Programmer

This section presents a layered description of an ODM system in an attempt to characterize the different interfaces and class libraries of an ODM system corresponding to the above roles. Each role represents use of the system at a different layer of abstraction.

Each layer of abstraction corresponds to one or more Class Libraries defining an interface through which certain objects and operations on those objects are visible. The Class Libraries define a vocabulary of messages that are understood by that interface.

The objects and operations visible at a given interface are implemented in terms of operations and objects defined by lower layers, since each layer provides the Extensibility attribute. The objects of the lower layers are not directly visible, since each layer preserves the Encapsulation attribute.

The users of each interface may define new objects (i.e. instances of some class visible at that interface), define new operations or methods, or define new classes (via extensibility).

Figure 2 on the next page shows a layered view of the roles and class libraries.

User Roles	**Class Libraries** (used by roles at this level
End Users, accesses the objects directly through ad-hoc query interfaces, and indirectly through other turn-key programs **Application Developer**, for building end-user applications **System Analyst**, for analyzing operations of the business, to perform tasks such as auditing, reporting, forecasting, and planning.	**Application Specific Class Libraries** **Enterprise or Business Class Library** These objects might implement an enterprise-wide schema which could be shared by a variety of applications.
Database Administrator, for system administration functions such as backup, tuning, etc. **Database Designer**, for defining the Enterprise Objects. **Database Method Programmer**, for implementing the operations or methods for the Enterprise Objects	**ODM System DL & Kernel Objects** The ODM sytem's Data Language, and Kernel Objects typically include the object equivalents of Relations and built-in datatypes of an RDBMS, as well as Objects or Classes which provide a Data Dictionary or System Catalog function.
ODM systems programmer, to implement Kernel Objects and Data Language, or write primitive methods such as access methods.	ODM System Internals, i.e. , Implementation Objects or Function Libraries
ODM vendor systems programmer, to implement the ODM System Internals.	Operating System run-time library and system programming languages Computer CPU, Storage, and Networking hardware

Figure 2 User Roles and Class Libraries

3.2 Application Program Interfaces

There are several possible configurations for connecting an application program to an ODM system. These configurations may be characterized in terms of:

* Language characteristics of the application program and/or the data language of the ODM system - How many languages are used to write the application? Is the language(s) object based?

* Level of automation of persistence - What is the distinction, if any, between transient and persistent objects? Does the application programmer have to write explicit statements to save persistent objects?

* Number of object spaces - How many different universes of object identifiers exist? Is there more than one "heap" of temporary and/or persistent objects?

* Number of operation execution spaces - How many execution spaces are there? If one space, is it in the ODM system or in the application programming language space? Are there execution spaces in both the ODM system and in the application program?

There are at least four possible configurations for using an ODM system . These alternatives are representative of currently available ODM technology.

1. Single data language with automatic persistence and a single object space. In this configuration, the language compiler and/or ODM system implement the automatic persistence.

2. Single language with semi-automatic or programmer-controlled persistence and two object spaces (transient objects and persistent objects)

3. Two object languages (application language, ODM data language) with semi-automatic or programmer controlled persistence and two object spaces (application space, database space)

4. Conventional language (such as C or Cobol) plus ODM data language, with programmer controlled persistence and two object spaces (application stack or heap, database)

Another dimension for characterizing an application program interface to an ODM system is whether structural access is allowed. Structural access involves simple, low-level messages to the database where the arguments and results of operations are very closely related to the state or information encapsulated inside of the receiver object. In the limit, structural access may reduce to navigational fetch and store operations which violate encapsulation.

Export Authority: 22 CFR 125.4 (b) (13)

OBJECT-ORIENTED DBMS REQUIREMENTS

by
Keith A. Marrs
Laila G. Robinson

McDonnell Douglas Corporation
McDonnell Aircraft Company
P.O. Box 516, Mail Code 0861020
St. Louis, Mo. 63166

INTRODUCTION

The information management requirements of a large, complex enterprise as the McDonnell Douglas Corporation (MDC) are best served in an environment wherein:

- Data sharing is enabled among organizations that need to share information.
- The addition of new systems and reorganization of existing systems are facilitated through the use of standards and open architectures.
- Navigational data access is hidden from applications so that physical databases can be revised without impacting application code.
- Application and data portability encourage reuse and lessen redundancy of software development.
- Data semantics across the enterprise are integrated so that there is one common understanding of the data.
- These semantics are captured and managed by intelligent data management systems so that they can be enforced consistently through these systems and data integrity is assured.

The progress towards achieving this has been impeded by various factors such as:

- The use of heterogeneous vendor data management products with proprietary interfaces.
- The inadequacy of data management products to capture and store data semantics.
- The lack of capabilities to access data from various heterogeneous databases distributed across the enterprise.

The addition of systems to the heterogeneous environment, particularly those using new technology such as object-oriented database management should alleviate instead of exacerbate this condition.

REQUIREMENTS

Object-Oriented database management systems (OODBMSs) have emerged as a way to capture and implement more data semantics than current DBMSs. OODBMSs can manage more

complicated data structures and operations, and they enable the enforcement of more rules on the data. Therefore we view OODBMSs as strong candidates in managing databases such as those residing in engineering and manufacturing which are heretofore managed by application code due to inadequacies of current DBMSs.

We have investigated the state of the art of OODBMSs, modeled several application data to determine their semantics, and we are developing a design for a data management prototype using object-oriented design methodologies. From these works we have defined a set of requirements for OODBMSs which will aid the achievement of the information management environment discussed above. These requirements were reviewed and approved by representatives from several component companies of MDC. They address several of the areas of interest for this workshop, including Object Data Model, Query and Data Manipulation Languages, OODB System Architecture and Interfaces, Change Management, and Transactions. The requirements are given below, listed by topic area.

Object Data Model

1. The object definition must include the following.

    - The object structures (i.e., the instance variables of the object).

    - The valid messages to the object.

    - The methods corresponding to the messages - these should be defined with a robust language.

    - The constraints on the object - these include constraints on the structures and methods, and constraints to enforce semantic integrity rules, including:

        - referential integrity which guarantees that every object referenced by another object exists in the database;

        - domain integrity which asserts that the value of an instance variable of an object must be a member of a set of eligible values (i.e., the domain) for that instance variable;

        - the handling of null values for object instance variables in methods and queries should correspond to the treatment of null values by ANSI standard SQL;

        - default values for object instance variables;

        - assertions; and

        - support for triggers such that the firing of these triggers is deterministic (i.e., predictable results).

    - The identifier of the object which is persistent and known to the user but which is not updatable and does not carry any semantics.

2. The following object features must be supported.

    - Encapsulation - only the interface (messages) of an object must be visible to a use

- Abstraction mechanisms, specifically:
  - classification which is the grouping of objects with common semantics into a class (i.e., a class defines the structures, messages, methods, and constraints for its group of objects--instances of the class; and because a class is also an object, it may define additional structures, messages, methods, and constraints which are used to identify properties and operations for the class itself or to store constant or default values for all the instances of the class);
  - generalization which is the grouping of similar classes into superclasses (i.e., all of the common structures, messages, methods, and constraints of the similar classes are associated with the superclass);
  - aggregation which is an abstraction in which a relationship between objects is regarded as a higher-level object (i.e., an aggregate object refers to other objects in its definition);
  - composition/dependent subparts which is an abstraction in which a group of objects are combined to form a composite object such that the existence of the component objects are dependent on the existence of the composite object; and
  - collections which are groups of objects that may be arbitrary or instances of the same class; these collections can be used for domains or to support ad hoc queries over all the instances of a class.
- Inheritance and multiple inheritance of properties, methods, constraints, and values. More specifically this includes:
  - class-class inheritance where a class inherits all the structures, messages, methods, and constraints of its superclasses;
  - class-instance inheritance where each instance of a class has values for the instance variables defined for the class and inherits the messages, methods, and constraints of the class; and
  - instance-instance inheritance where one instance inherits the value of an instance variable and the messages, methods, and constraints associated with this instance variable from another instance; this is often used for composite objects to allow component objects to assume certain features of the composite object (e.g., the color of a car door is inherited from the color of the car).
- Extensibility, specifically:
  - a base set of objects (e.g., integers and dates), which are extensible and can be used to construct other objects, must be provided; and
  - users must be allowed to define new structures, operations, and access methods to handle application entities.
- Name conflict resolution at all levels such that a user is notified of conflicts during schema definition.
- Aliases/synonyms for objects and object instance variables should be supported. This will reduce name scopes, avoid possible name conflicts, and allow context dependent names.

3. The following types of objects must be supported.

   - Recursive objects which are objects that are composed of other objects of the same class.

   - Spatial objects which are objects whose position and area/volume in 2- or 3-D space needs to be supported. Queries such as how near one object is to another object, or what objects are contained within some area must be supported for these objects.

   - Temporal objects which are objects whose relevant information depends on time.

   - Large objects such as maps and schematic layouts.

   - Multimedia objects such as 3-D graphics, text, and image.

   - Temporary objects which are objects needed for some transaction or application but do not need to persist.

Query and Data Manipulation Languages

ANSI SQL with appropriate object-oriented extensions should be used as both the data definition and data manipulation language. One appropriate extension is to allow object methods to be invoked from a SQL statement.

OODB System Architecture and Interfaces

1. The OODBMS must be able to be integrated with other DBMSs by means of ANSI standard SQL. This implies an open architecture, interoperability, and distributed operations across DBMSs.

2. The three-schema architecture must be supported. More specifically, this must include support for the following.

   - Derived objects (views) - this includes mapping between external objects and conceptual objects, and mapping between conceptual objects and internal objects. Creation, manipulation, and access operation must be supported for these derived objects.

   - User-specified external object presentation formats - these should be stored and persistent in the database.

   - Information for the persistent objects (e.g., class information) at each level of the three-schema architecture, and the schema mappings must be stored in the data dictionary/directory and automatically managed. This includes automatic execution of mappings between the schemas during data access and manipulation.

3. Participation in a distributed DBMS must be supported.

   - Initially, the OODBMS should run on the client/server architecture using the ANSI/ISO RDA standards.

- Access to relational DBMSs must be supported. In this situation, the object-oriented DBMS could be both client and server.

- To support replicated data and multi-user access within the distributed DBMS environment, the distributed system must support:

    - snapshots of objects; and

    - deferred and immediate updates of copies of objects.

- A true heterogeneous distributed architecture must be supported in the future. It must include:

    - management of a global conceptual schema, that is, a conceptual schema that applies to all of the participants in the distributed system;

    - a distributed database manager;

    - support of other vendor's DBMSs; and

    - user transparencies to DBMS heterogeneity (i.e., the distributed nature of the DBMS should be transparent to the user).

4. A single data dictionary/directory facility must be supported. More specifically,

    - It must be based on ANSI IRDS standards.

    - It must be an active data dictionary. The data dictionary should be an integral part of constraint enforcement and three-schema mapping at run time.

    - It must be a dynamic data dictionary. The data dictionary can be updated at run time.

    - It must be loosely coupled with the DBMS such that it can interface with other DBMSs.

    - Schema modification must be allowed - this includes modifying or extending object definitions.

    - A browser must be provided for interactively examining the data dictionary.

5. Archiving/restoring of objects to cheaper storage devices must be supported. This includes storing immutable objects on CD-ROM devices.

6. The following language interfaces and characteristics must be supported.

    - ANSI SQL with appropriate object-oriented extensions should be used as both the data definition and data manipulation language. One appropriate extension is to allow object methods to be invoked from a SQL statement.

    - Host language interfaces, including embedded and module interfaces. An embedded interface is one in which SQL statements can be mixed with the host language statements. A module interface is one in which separate modules are defined for

interacting with the database and these modules can be called from the host language.

- A 4GL tool that can provide access across DBMSs and interfaces with ANSI standard SQL.

- A graphics interface for schema definition and management.

- Access to multiple levels of a composite object in the same query must be supported. This capability allows a part-of hierarchy to be examined in one query.

- Late, or dynamic, binding of objects and methods, that is binding at runtime instead of compile time. This will reduce code maintenance due to schema changes, such as changes in the class hierarchy.

7. Portability of applications and the DBMS across hardware and operating system platforms must be supported.

8. Data modeling must be supported. (IDEF, NIAM, EXPRESS, IDEF1x - extensions for objects) These tools should interface to the DBMS through the data dictionary/directory of the DBMS.

9. Prototyping tools must be supported. These tools should provide mechanisms for schema creation and management, and for populating a database.

Change Management

Configuration management must be supported, this includes support for versions of objects and versions of schema.

Transactions

Transaction management facilities for the following types of transactions must be supported.

- Long transactions in which objects may be checked out for hours or days.

- Stored transactions.

- Nested transactions.

- Interactive, on-line transactions.

- Partial rollbacks in which a transaction is rolled back to the last savepoint defined in the transaction.

In addition to all the above, the following traditional DBMS features must be supported.

- Multi-user environment.

- Backup and recovery mechanisms to handle crashes.
- Concurrency control mechanisms to allow concurrent sharing of objects.
- Authorization and security mechanisms where the object is the lowest level of granularity.
- Authorization and security mechanisms where the properties of an object are the lowest level of granularity.
- Privileges on the access and manipulation of objects and object properties.
- Performance tuning features, including clustering of objects, indexing, and user-defined access methods.
- Import and export (often between DBMSs) facilities for objects.
- Interactive sessions and batch processing.

CONCLUSION

For OODBMSs to be useful in large complex enterprises, they must be able to participate and integrate into a data sharing, heterogeneous information management environment. The adoption of standards is a major mechanism for this integration. To reduce additional software development and maintenance by end users when integrating this new technology, extension of existing standards must be given the highest priority. New standards should then be created for the remaining areas, such as the Object Data Model, which are not covered by the extended standards.

THIRD-GENERATION DATA BASE SYSTEM MANIFESTO

The Committee for Advanced DBMS Function[1]

Abstract

We call the older hierarchical and network systems **first generation** database systems and refer to the current collection of relational systems as the **second generation**. In this paper we consider the characteristics that must be satisfied by the next generation of data managers, which we call **third generation** database systems.

Our requirements are collected into three basis tenets along with 13 more detailed propositions.

1. INTRODUCTION

The network and hierarchical database systems that were prevalent in the 1970's are aptly classified as **first generation** database systems because they were the first systems to offer substantial DBMS function in a unified system with a data definition and data manipulation language for collections of records.[2] CODASYL systems [CODA71] and IMS [DATE86] typify such first generation systems.

In the 1980's first generation systems were largely supplanted by the current collection of relational DBMSs which we term **second generation** database systems. These are widely believed to be a substantial step forward for many applications over first generation systems because of their use of a non-procedural data manipulation language and their provision of a substantial degree of data independence. Second generation systems are typified by DB2, INGRES, NON-STOP SQL, ORACLE and Rdb/VMS.[3]

However, second generation systems were focused on **business data processing** applications, and many researchers have pointed out that they are inadequate for a broader class of applications. Computer aided design (CAD), computer aided software engineering (CASE) and hypertext applications are often singled out as examples that could effectively utilize a different kind of DBMS with specialized capabilities. Consider, for example, a publishing application in which a client wishes to arrange the layout of a newspaper and then print it. This application requires storing text segments, graphics, icons, and the myriad of other kinds of data elements found in most hypertext environments. Supporting such data elements is usually difficult in second generation systems.

However, critics of the relational model fail to realize a crucial fact. Second generation systems do not support **most** business data processing applications all that well. For example, consider an insurance application that processes claims. This application requires traditional data elements such as the name and coverage of each person insured. However, it is desirable to store images of photographs of the event to

---

[1]The committee is composed of Michael Stonebraker of the University of California, Berkeley, Lawrence A. Rowe of the University of California, Berkeley, Bruce Lindsay of IBM Research, James Gray of Tandem Computers, Michael Carey of the University of Wisconsin, Michael Brodie of GTE Laboratories, Philip Bernstein of Digital Equipment Corporation, and David Beech of Oracle Corporation.

[2]To discuss relational and other systems without confusion, we will use neutral terms in this paper. Therefore, we define a **data element** as an atomic data value that is stored in the database. Every data element has a data type (or type for short), and data elements can be assembled into a **record** which is a set of one or more named data elements. Lastly, a **collection** is a named set of records, each with the same number and type of data elements.

[3]DB2, INGRES, NON-STOP SQL, ORACLE and Rdb/VMS are trademarks respectively of IBM, INGRES Corporation, Tandem, ORACLE Corporation, and Digital Equipment Corporation.

which a claim is related as well as a facsimile of the original hand-written claim form. Such data elements are also difficult to store in second generation DBMSs. Moreover, all information related to a specific claim is aggregated into a **folder** which contains traditional data, images and perhaps procedural data as well. A folder is often very complex and makes the data elements and aggregates of CAD and CASE systems seem fairly routine by comparison.

Thus, almost everybody requires a better DBMS, and there have been several efforts to construct prototypes with advanced function. Moreover, most current DBMS vendors are working on major functional enhancements of their second generation DBMSs. There is a surprising degree of consensus on the desired capabilities of these next-generation systems, which we term **third generation** database systems. In this paper, we present the three basic tenets that should guide the development of third generation systems. In addition, we indicate 13 propositions which discuss more detailed requirements for such systems. Our paper should be contrasted with those of [ATKI89, KIM90, ZDON90] which suggest different sets of tenets.

2. THE TENETS OF THIRD-GENERATION DBMSs

The first tenet deals with the definition of third generation DBMSs.

TENET 1: Besides traditional data management services, third generation DBMSs will provide support for richer object structures and rules.

Data management characterizes the things that current relational systems do well, such as processing 100 transactions per second from 1000 on-line terminals and efficiently executing six way joins. Richer object structures characterize the capabilities required to store and manipulate non-traditional data elements such as text and spatial data. In addition, an application designer should be given the capability of specifying a set of rules about data elements, records and collections.[4] Referential integrity in a relational context is one simple example of such a rule; however, there are many more complex ones.

We now consider two simple examples that illustrate this tenet. Return to the newspaper application described earlier. It contains many non-traditional data elements such as text, icons, maps, and advertisement copy; hence richer object structures are clearly required. Furthermore, consider the classified advertisements for the paper. Besides the text for the advertisement, there are a collection of business data processing data elements, such as the rate, the number of days the advertisement will run, the classification, the billing address, etc. Any automatic newspaper layout program requires access to this data to decide whether to place any particular advertisement in the current newspaper. Moreover, selling classified advertisements in a large newspaper is a standard transaction processing application which requires traditional data management services. In addition, there are many rules that control the layout of a newspaper. For example, one cannot put an advertisement for Macy's on the same page as an advertisement for Nordstrom. The move toward semi-automatic or automatic layout requires capturing and then enforcing such rules. As a result there is need for rule management in our example application as well.

Consider next our insurance example. As noted earlier, there is the requirement for storing non-traditional data elements such as photographs and claims. Moreover, making changes to the insurance coverage for customers is a standard transaction processing application. In addition, an insurance application requires a large collection of rules such as

> Cancel the coverage of any customer who has had a claim of type Y over value X.
> Escalate any claim that is more than N days old.

We have briefly considered two applications and demonstrated that a DBMS must have data, object and rules services to successfully solve each problem. Although it is certainly possible that niche markets will be available to systems with lesser capabilities, the successful DBMSs of the 90's will have services in all three areas.

---

[4]See the previous footnote for definitions of these terms.

We now turn to our second fundamental tenet.

TENET 2: Third generation DBMSs must subsume second generation DBMSs.

Put differently, second generation systems made a major contribution in two areas:

non-procedural access
data independence

and these advances must not be compromised by third generation systems.

Some argue that there are applications which **never** wish to run queries because of the simplicity of their DBMS accesses. CAD is often suggested as an example with this characteristic [CHAN89]. Therefore, some suggest that future systems will not require a query language and consequently do not need to subsume second generation systems. Several of the authors of this paper have talked to **numerous** CAD application designers with an interest in databases, and all have specified a query language as a necessity. For example, consider a mechanical CAD system which stores the parts which compose a product such as an automobile. Along with the spatial geometry of each part, a CAD system must store a collection of **attribute** data, such as the cost of the part, the color of the part, the mean time to failure, the supplier of the part, etc. CAD applications require a query language to specify ad-hoc queries on the attribute data such as:

How much does the cost of my automobile increase if supplier X raises his prices by Y percent?

Consequently, we are led to a query language as an absolute requirement.

The second advance of second generation systems was the notion of data independence. In the area of physical data independence, second generation systems automatically maintain the consistency of all access paths to data, and a query optimizer automatically chooses the best way to execute any given user command. In addition, second generation systems provide **views** whereby a user can be insulated from changes to the underlying set of collections stored in the database. These characteristics have dramatically lowered the amount of program maintenance that must be done by applications and should not be abandoned.

Tenet 3 discusses the final philosophical premise which must guide third generation DBMSs.

TENET 3: Third generation DBMSs must be open to other subsystems.

Stated in different terms, any DBMS which expects broad applicability must have a fourth generation language (4GL), various decision support tools, friendly access from many programming languages, friendly access to popular subsystems such as LOTUS 1-2-3, interfaces to business graphics packages, the ability to run the application on a different machine from the database, and a distributed DBMS. All tools and the DBMS must run effectively on a wide variety of hardware platforms and operating systems.

This fact has two implications. First, any successful third generation system must support most of the tools described above. Second, a third generation DBMS must be **open**, i.e. it must allow access from additional tools running in a variety of environments. Moreover, each third generation system must be willing to participate with other third generation DBMSs in future distributed database systems.

These three tenets lead to a variety of more detailed propositions on which we now focus.

3. THE THIRTEEN PROPOSITIONS

There are three groups of detailed propositions which we feel must be followed by the successful third generation database systems of the 1990s. The first group discusses propositions which result from Tenet 1 and refine the requirements of object and rule management. The second group contains a collection of propositions which follow from the requirement that third generation DBMSs subsume second generation ones. Finally, we treat propositions which result from the requirement that a third generation system be open.

3.1. Propositions Concerning Object and Rule Management

DBMSs cannot possibly anticipate all the kinds of data elements that an application might want. Most people think, for example, that time is measured in seconds and days. However, all months have 30 days in bond trading applications, the day ends at 15:30 for most banks, and "yesterday" skips over week-ends and holidays for stock market applications. Hence, it is imperative that a third generation DBMS manage a diversity of objects and we have 4 propositions that deal with object management and consider type constructors, inheritance, functions and unique identifiers.

PROPOSITION 1.1: A third generation DBMS must have a rich type system.

All of the following are desirable:

1) an abstract data type system to construct new base types
2) an array type constructor
3) a sequence type constructor
4) a record type constructor
5) a set type constructor
6) functions as a type
7) a union type constructor
8) recursive composition of the above constructors

The first mechanism allows one to construct new base types in addition to the standard integers, floats and character strings available in most systems. These include bit strings, points, lines, complex numbers, etc. The second mechanism allows one to have arrays of data elements, such as found in many scientific applications. Arrays normally have the property that a new element cannot be inserted into the middle of the array and cause all the subsequent members to have their position incremented. In some applications such as the lines of text in a document, one requires this insertion property, and the third type constructor supports such sequences. The fourth mechanism allows one to group data elements into records. Using this type constructor one could form, for example, a record of data items for a person who is one of the "old guard" of a particular university. The fifth mechanism is required to form unordered collections of data elements or records. For example, the set type constructor is required to form the set of all the old guard. We discuss the sixth mechanism, functions (methods) in Proposition 1.3; hence, it is desirable to have a DBMS which naturally stores such constructs. The next mechanism allows one to construct a data element which can take a value from one of several types. Examples of the utility of this construct are presented in [COPE84]. The last mechanism allows type constructors to be recursively composed to support complex objects which have internal structure such as documents, spatial geometries, etc. Moreover, there is no requirement that the last type constructor applied be the one which forms sets, as is true for second generation systems.

Besides implementing these type constructors, a DBMS must also extend the underlying query language with appropriate constructs. Consider, for example, the SALESPERSON collection, in which each salesperson has a name and a quota which is an array of 12 integers. In this case, one would like to be able to request the names of salespersons with April quotas over $5000 as follows:

```
select name
from SALESPERSON
where quota[4] > 5000
```

Consequently, the query language must be extended with syntax for addressing into arrays. Prototype syntax for a variety of type constructors is contained in [CARE88].

The utility of these type constructors is well understood by DBMS clients who have data to store with a richer structure. Moreover, such type constructors will also make it easier to implement the persistent programming languages discussed in Proposition 3.2. Furthermore, as time unfolds it is certainly possible that additional type constructors may become desirable. For example, transaction processing systems manage queues of messages [BERN90]. Hence, it may be desirable to have a type constructor which forms queues.

Second generation systems have few of these type constructors, and the advocates of Object-oriented Data Bases (OODB) claim that entirely new DBMSs must come into existence to support these features. In this regard, we wish to take strong exception. There are prototypes that demonstrate how to add many of

the above type constructors to relational systems. For example, [STON83] shows how to add sequences of records to a relational system, [ZANI83] and [DADA86] indicate how to construct certain complex objects, and [OSBO86, STON86] show how to include an ADT system. We claim that all these type constructors can be added to relational systems as natural enhancements and that the technology is relatively well understood.[5] Moreover, commercial relational systems with some of these features have already started to appear.

Our second object management proposition concerns inheritance.

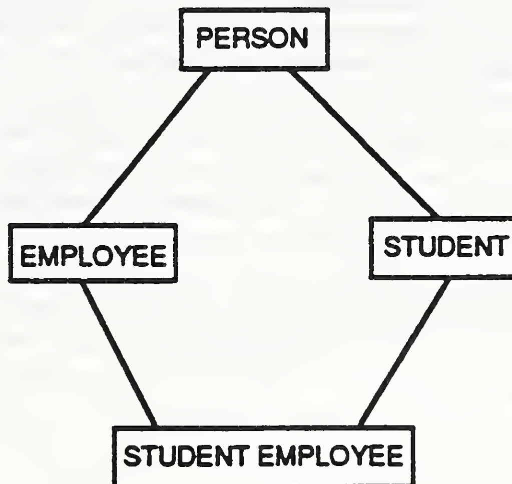
PROPOSITION 1.2: Inheritance is a good idea.

Much has been said about this construct, and we feel we can be very brief. Allowing types to be organized into an inheritance hierarchy is a good idea. Moreover, we feel that multiple inheritance is essential, so the inheritance hierarchy must be a directed graph. If only single inheritance is supported, then we feel that there are too many situations that cannot be adequately modeled. For example, consider a collection of instances of PERSON. There are two specializations of the PERSON type, namely STUDENT and EMPLOYEE. Lastly, there is a STUDENT EMPLOYEE, which should inherit from both STUDENT and EMPLOYEE. In each collection, data items appropriate to the collection would be specified when the collection was defined and others would be inherited from the parent collections. A diagram of this situation, which demands multiple inheritance, is indicated in Figure 1. While [ATKI89] advocates inheritance, it lists multiple inheritance as an optional feature.

Moreover, it is also desirable to have collections which specify no additional fields. For example, TEENAGER might be a collection having the same data elements as PERSON, but having a restriction on ages. Again, there have been prototype demonstrations on how to add these features to relational systems, and we expect commercial relational systems to move in this direction.

Our third proposition concerns the inclusion of functions in a third generation DBMS.

PROPOSITION 1.3: Functions, including database procedures and methods, and



A Typical Multiple Inheritance Hierarchy
Figure 1

[5] One might argue that a relational system with all these extensions can no longer be considered "relational", but that is not the point. The point is that such extensions are possible and quite natural.

encapsulation are a good idea.

Second generation systems support functions and encapsulation in restricted ways. For example, the operations available for tables in SQL are implemented by the functions **create, alter,** and **drop.** Hence, the table abstraction is only available by executing one of the above functions.

Obviously, the benefits of encapsulation should be made available to application designers so they can associate functions with user collections. For example, the functions HIRE(EMPLOYEE), FIRE(EMPLOYEE) and RAISE-SAL(EMPLOYEE) should be associated with the familiar EMPLOYEE collection. If users are not allowed direct access to the EMPLOYEE collection but are given these functions instead, then all knowledge of the internal structure of the EMPLOYEE collection is encapsulated within these functions.

Encapsulation has administrative advantages by encouraging modularity and by registering functions along with the data they encapsulate. If the EMPLOYEE collection changes in such a way that its previous contents cannot be defined as a view, then all the code which must be changed is localized in one place, and will therefore be easier to change.

Encapsulation often has performance advantages in a protected or distributed system. For example, the function HIRE(EMPLOYEE) may make a number of accesses to the database while executing. If it is specified as a function to be executed internally by the data manager, then only one round trip message between the application and the DBMS is executed. On the other hand, if the function runs in the user program then one round trip message will be executed for each access. Moving functions inside the DBMS has been shown to improve performance on the popular Debit-Credit benchmark [ANON85].

Lastly, such functions can be inherited and possibly overridden down the inheritance hierarchy. Therefore, the function HIRE(EMPLOYEE) can automatically be applied to the STUDENT EMPLOYEE collection. With overriding, the implementation of the function HIRE can be rewritten for the for the STUDENT EMPLOYEE collection. In summary, encapsulated functions have performance and structuring benefits and are highly desirable. However, there are three comments which we must make concerning functions.

First, we feel that users should write functions in a higher level language (HLL) and obtain DBMS access through a high-level non-procedural access language. This language may be available through an embedding via a preprocessor or through direct extension of the HLL itself. Put differently, functions should run queries and not perform their own navigation using calls to some lower level DBMS interface. Proposition 2.1 will discuss the undesirability of constructing user programs with low-level data access interfaces, and the same discussion applies equally to the construction of functions.

There are occasional requirements for a function to directly access internal interfaces of a DBMS. This will require violating our admonition above about only accessing the database through the query language, and an example of such a function is presented in [STON90]. Consequently, direct access to system internals should probably be an allowable but highly discouraged (!) way to write functions.

Our second comment concerns the notion of **opaque types.** Some OODB enthusiasts claim that the only way that a user should be able to access a collection is to execute some function available for the collection. For example, the only way to access the EMPLOYEE collection would be to execute a function such as HIRE(EMPLOYEE). Such a restriction ignores the needs of the query language whose execution engine requires access to each data element directly. Consider, for example:

```
select *
from EMPLOYEE
where salary > 10000
```

To solve this query, the execution engine must have direct access to the salary data elements and any auxiliary access paths (indexes) available for them. Therefore, we believe that a mechanism is required to makes types **transparent,** so that data elements inside them can be accessed through the query language. It is possible that this can be accomplished through an automatically defined "accessor" function for each data element or through some other means. An authorization system is obviously required to control access to the database through the query language.

Our last comment concerns the commercial marketplace. All major vendors of second generation DBMSs already support functions coded in a HLL (usually the 4GL supported by the vendor) that can

make DBMS calls in SQL. Moreover, such functions can be used to encapsulate accesses to the data they manage. Hence, functions stored in the database with DBMS calls in the query language are already commonplace commercially. The work remaining for the commercial relational vendors to support this proposition is to allow inheritance of functions. Again there have been several prototypes which show that this is a relatively straightforward extension to a relational DBMS. Yet again, we see a clear path by which current relational systems can move towards satisfying this proposition.

Our last object management proposition deals with the automatic assignment of unique identifiers.

PROPOSITION 1.4: Unique Identifiers (UIDs) for records should be assigned by the DBMS only if a user-defined primary key is not available.

Second generation systems support the notion of a **primary key**, which is a user-assigned unique identifier. If a primary key exists for a collection that is known **never** to change, for example social security number, student registration number, or employee number, then no additional system-assigned UID is required. An **immutable** primary key has an extra advantage over a system-assigned unique identifier because it has a natural, human readable meaning. Consequently, in data interchange or debugging this may be an advantage.

If no primary key is available for a collection, then it is imperative that a system-assigned UID be provided. Because SQL supports update through a cursor, second generation systems must be able to update the last record retrieved, and this is only possible if it can be uniquely identified. If no primary key serves this purpose, the system must include an extra UID. Therefore, several second generation systems already obey this proposition.

Moreover, as will be noted in Proposition 2.3, some collections, e.g. views, do not necessarily have system assigned UIDs, so building a system that requires them is likely to be proven undesirable. We close our discussion on Tenet 1 with a final proposition that deals with the notion of rules.

PROPOSITION 1.5: Rules (triggers, constraints) will become a major feature in future systems. They should not be associated with a specific function or collection.

OODB researchers have generally ignored the importance of rules, in spite of the pioneering use of active data values and daemons in some programming languages utilizing object concepts. When questioned about rules, most OODB enthusiasts either are silent or suggest that rules be implemented by including code to support them in one or more functions that operate on a collection. For example, if one has a rule that every employee must earn a smaller salary than his manager, then code appropriate to this constraint would be inserted into both the HIRE(EMPLOYEE) and the RAISE-SAL(EMPLOYEE) functions.

There are two fundamental problems with associating rules with functions. First, whenever a new function is added, such as PENSION-CHANGE(EMPLOYEE), then one must ensure that the function in turn calls RAISE-SAL(EMPLOYEE), or one must include code for the rule in the new function. There is no way to guarantee that a programmer does either; consequently, there is no way to guarantee rule enforcement. Moreover, code for the rule must be placed in at least two functions, HIRE(EMPLOYEE) and RAISE-SAL(EMPLOYEE). This requires duplication of effort and will make changing the rule at some future time more difficult.

Next, consider the following rule:

Whenever Joe gets a salary adjustment, propagate the change to Sam.

Under the OODB scheme, one must add appropriate code to both the HIRE and the RAISE-SAL functions. Now suppose a second rule is added:

Whenever Sam gets a salary adjustment, propagate the change to Fred.

This rule will require inserting additional code into the same functions. Moreover, since the two rules interact with each other, the writer of the code for the second rule must understand all the rules that appear in the function he is modifying so he can correctly deal with the interactions. The same problem arises when a rule is subsequently deleted.

Lastly, it would be valuable if users could ask queries about the rules currently being enforced. If they are buried in functions, there is no easy way to do this.

In our opinion there is only one reasonable solution; rules must be enforced by the DBMS but not bound to any function or collection. This has two consequences. First, the OODB paradigm of "everything is expressed as a method" simply does not apply to rules. Second, one cannot directly access any internal interfaces in the DBMS below the rule activation code, which would allow a user to bypass the run time system that wakes up rules at the correct time.

In closing, there are already products from second generation commercial vendors which are faithful to the above proposition. Hence, the commercial relational marketplace is ahead of OODB thinking concerning this particular proposition.

3.2. Propositions Concerning Increasing DBMS Function

We claimed earlier that third generation systems could not take a step backwards, i.e. they must subsume all the capabilities of second generation systems. The capabilities of concern are query languages, the specification of sets of data elements and data independence. We have four propositions in this section that deal with these matters.

PROPOSITION 2.1: All programatic access to a database should be through a non-procedural, high-level access language.

Much of the OODB literature has underestimated the critical importance of high-level data access languages with expressive power equivalent to a relational query language. For example, [ATKI89] proposes that the DBMS offer an ad hoc query facility in any convenient form. We make a much stronger statement: the expressive power of a query language must be present in every programmatic interface and it is the only way to access DBMS data. Long term, this service can be provided by adding query language constructs to the multiple persistent programming languages that we discuss further in Proposition 3.2. Short term, this service can be provided by embedding a query language in conventional programming languages.

Second generation systems have demonstrated that dramatically lower program maintenance costs result from using this approach relative to first generation systems. In our opinion, third generation database systems **must not** compromise this advance. By contrast, many OODB researchers state that the applications for which they are designing their systems wish to **navigate** to desired data using a low-level procedural interface. Specifically, they want an interface to a DBMS in which they can access a specific record. One or more data elements in this record would be of type "reference to a record in some other collection" typically represented by some sort of pointer to this other record, e.g an object identifier. Then, the application would dereference one of these pointers to establish a new current record. This process would be repeated until the application had navigated to the desired records.

This navigational point of view is well articulated in the Turing Award presentation by Charles Bachman [BACH73]. We feel that the subsequent 17 years of history has demonstrated that this kind of interface is undesirable and should not be used. Here we summarize only two of the more important problems with navigation. First, when the programmer navigates to desired data in this fashion, he is replacing the function of the query optimizer by hand-coded lower level calls. It has been clearly demonstrated by history that a well-written, well-tuned, optimizer can almost always do better than a programmer can do by hand. Hence, the programmer will produce a program which has inferior performance. Moreover, the programmer must be considerably smarter to code against a more complex lower level interface.

However, the real killer concerns **schema evolution.** If the number of indexes changes or the data is reorganized to be differently clustered, there is no way for the navigation interface to automatically take advantage of such changes. Hence, if the physical access paths to data change, then a programmer must modify his program. On the other hand, a query optimizer simply produces a new plan which is optimized for the new environment. Moreover, if there is a change in the collections that are physically stored, then the support for **views** prevalent in second generation systems can be used to insulate the application from the change. To avoid these problems of schema evolution and required optimization of database access in each program, a user should specify the set of data elements in which he is interested as a query in a non-procedural language.

However, consider a user who is **browsing** the database, i.e. navigating from one record to another. Such a user wishes to see all the records on any path through the database that he explores. Moreover, which path he examines next may depend on the composition of the current record. Such a user is clearly accessing a single record at a time algorithmically. Our position on such users is straight-forward, namely they should run a sequence of queries that return a single record, such as:

```
select *
from collection
where collection.key = value
```

Although there is little room for optimization of such queries, one is still insulated from required program maintenance in the event that the schema changes. One does not obtain this service if a lower level interface is used, such as:

```
dereference (pointer)
```

Moreover, we claim that our approach yields comparable performance to that available from a lower level interface. This perhaps counter-intuitive assertion deserves some explanation. The vast majority of current OODB enthusiasts suggest that a pointer be **soft**, i.e. that its value not change even if the data element that it points to is moved. This characteristic, **location independence,** is desirable because it allows data elements to be moved without compromising the structure of the database. Such data element movement is often inevitable during database reorganization or during crash recovery. Therefore, OODB enthusiasts recommend that location independent unique identifiers be used for pointers. As a result, dereferencing a pointer requires an access to a hashed or indexed structure of unique identifiers.

In the SQL representation, the pair:

(relation-name, key)

is exactly a location independent unique identifier which entails the same kind of hashed or indexed lookup. Any overhead associated with the SQL syntax will presumably be removed at compile time.

Therefore we claim that there is little, if any, performance benefit to using the lower level interface when a single data element is returned. On the other hand, if multiple data elements are returned then replacing a high level query with multiple lower level calls may degrade performance, because of the cost of those multiple calls from the application to the DBMS.

The last claim that is often asserted by OODB enthusiasts is that programmers, e.g. CAD programmers, want to perform their own navigation, and therefore, a system should encourage navigation with a low-level interface. We recognize that certain programmers probably prefer navigation. There were programmers who resisted the move from assembly language to higher level programming languages and others who resisted moving to relational systems because they would have a less complex task to do and therefore a less interesting job. Moreover, they thought they could do a better job than compilers and optimizers. We feel that the arguments against navigation are compelling and that some programmers simply require education.

Therefore, we are led to conclude that all DBMS access should be specified by queries in a non-procedural high-level access notation. In Proposition 3.2 we will discuss issues of integrating such queries with current HLLs.

We now turn to a second topic for which we believe that a step backwards must also be avoided. Third generation systems will support a variety of type constructors for collections as noted in Proposition 1.1, and our next proposition deals with the specification of such collections, especially collections which are sets.

PROPOSITION 2.2: There should be at least two ways to specify collections, one using enumeration of members and one using the query language to specify membership.

The OODB literature suggests specifying sets by enumerating the members of a set, typically by means of a linked list or array of identifiers for members [DEWI90]. We believe that this specification is generally an inferior choice. To explore our reasoning, consider the following example.

ALUMNI (name, age, address)
GROUPS (g-name, composition)

Here we have a collection of alumni for a particular university along with a collection of groups of alumni. Each group has a name, e.g. old guard, young turks, elders, etc. and the composition field indicates the alumni who are members of each of these groups. It is clearly possible to specify composition as an array of pointers to qualifying alumni. However, this specification will be quite inefficient because the sets in this example are likely to be quite large and have substantial overlap. More seriously, when a new person is added to the ALUMNI collection, it is the responsibility of the application programmer to add the new person to all the appropriate groups. In other words, the various sets of alumni are specified **extensionally** by enumerating their members, and membership in any set is **manually** determined by the application programmer.

On the other hand, it is also possible to represent GROUPS as follows:

GROUPS(g-name, min-age, max-age, composition)

Here, composition is specified **intensionally** by the following SQL expression:

```
select *
from ALUMNI
where age > GROUPS.min-age and age < GROUPS.max-age
```

In this specification, there is one query for each group, parameterized by the age requirement for the group. Not only is this a more compact specification for the various sets, but also it has the advantage that set membership is **automatic**. Hence, whenever a new alumnus is added to the database, he is automatically placed in the appropriate sets. Such sets are guaranteed to be semantically consistent.

Besides assured consistency, there is one further advantage of automatic sets, namely they have a possible performance advantage over manual sets. Suppose the user asks a query such as:

```
select g-name
from GROUPS
where composition.name = "Bill"
```

This query requests the groups in which Bill is a member and uses the "nested dot" notation popularized by GEM [ZANI83] to address into the members of a set. If an array of pointers specification is used for composition, the query optimizer may sequentially scan all records in GROUPS and then dereference each pointer looking for Bill. Alternately, it might look up the identifier for Bill, and then scan all composition fields looking for the identifier. On the other hand, if the intensional representation is used, then the above query can be transformed by the query optimizer into:

```
select g-name
from GROUPS, ALUMNI
where ALUMNI.name = "Bill"
and ALUMNI.age > GROUPS.min-age and ALUMNI.age < GROUPS.max-age
```

If there is an index on GROUPS.min-age or GROUPS.max-age and on ALUMNI.name, this query may substantially outperform either of the previous query plans.

In summary, there are at least two ways to specify collections such as sets, arrays, sequences, etc. They can be specified either **extensionally** through collections of pointers, or intensionally through expressions. Intensional specification maintains **automatic** set membership [CODA71], which is desirable in most applications. Extensional specifications are desirable only when there is no structural connection between the set members or when automatic membership is not desired.

Also with an intensional specification, semantic transformations can be performed by the optimizer, which is then free to use whatever access path is best for a given query, rather than being limited in any way by pointer structures. Hence, physical representation decisions can be delegated to the DBA where they belong. He can decide what access paths to maintain, such as linked lists or pointer arrays [CARE90].

Our point of view is that both representations are required, and that intensional representation should be favored. On the other hand, OODB enthusiasts typically recommend only extensional techniques. It should be pointed out that there was considerable attention dedicated in the mid 1970's to the advantages of automatic sets relative to manual sets [CODD74]. In order to avoid a step backwards, third generation systems must favor automatic sets.

Our third proposition in this section concerns **views** and their crucial role in database applications.

PROPOSITION 2.3: Updatable views are essential.

We see very few static databases; rather, most are dynamic and ever changing. In such a scenario, whenever the set of collections changes, then program maintenance may be required. Clearly, the encapsulation of database access into functions and the encapsulation of functions with a single collection is a helpful step. This will allow the functions which must be changed to be easily identified. However, this solution, by itself, is inadequate. If a change is made to the schema it may take weeks or even months to rewrite the affected functions. During this intervening time the database cannot simply be "down". Moreover, if changes occur rapidly, the resources consumed may be unjustifiable.

A clearly better approach is to support **virtual collections** (views). Second generation systems were an advance over first generation systems in part because they provided some support in this area. Unfortunately, it is often not possible to update relational views. Consequently, if a user performs a schema modification and then defines his previous collections as views, application programs which previously ran may or may not continue to do so. Third generation systems will have to do a better job on updatable views.

The traditional way to support view updates is to perform command transformations along the lines of [STON75]. To disambiguate view updates, additional semantic information must be provided by the definer of the view. One approach is to require that each collection be opaque which might become a view at a later time. In this case there is a group of functions through which all accesses to the collection are funneled [ROWE79], and the view definer must perform program maintenance on each of these functions. This will entail substantial program maintenance as well as disallow updates through the query language. Alternately, it has been shown [STON90B] that a suitable rules system can be used to provide the necessary semantics. This approach has the advantage that only one (or a small number) of rules need be specified to provide view update semantics. This will be simpler than changing the code in a collection of functions.

Notice that the members of a virtual collection do not necessarily have a unique identifier because they do not physically exist. Hence, it will be difficult to require that each record in a collection have a unique identifier, as dictated in many current OODB prototypes.

Our last point is that data independence cannot be given up, which requires that all physical details must be hidden from application programmers.

PROPOSITION 2.4: Performance indicators have almost nothing to do with data models and must not appear in them.

In general, the main determiners of performance using either the SQL or lower level specification are:

the amount of performance tuning done on the DBMS
the usage of compilation techniques by the DBMS
the location of the buffer pool (in the client or DBMS address space)
the kind of indexing available
the performance of the client-DBMS interface
and the clustering that is performed.

Such issues have nothing to do with the data model or with the usage of a higher level language like SQL versus a lower level navigational interface. For example, the tactic of clustering related objects together has been highlighted as an important OODB feature. However, this tactic has been used by data base systems for many years, and is a central notion in most IMS access methods. Hence, it is a physical representation issue that has nothing to do with the data model of a DBMS. Similarly, whether or not a system builds indexes on unique identifiers and buffers database records on a client machine or even in user space of an application program are not data model issues.

We have also talked to numerous programmers who are doing non traditional problems such as CAD, and are convinced that they require a DBMS that will support their application which is optimized for their environment. Providing subsecond response time to an engineer adding a line to an engineering drawing may require one or more of the following:

an access method for spatial data such as R-trees, hb-trees or grid files
a buffer pool on the engineer's workstation as opposed to a central server
a buffer pool in his application program
data buffered in screen format rather than DBMS format

These are all performance issues for a workstation/server environment and have nothing to do with the data model or with the presence or absence of a navigational interface.

For a given workload and database, one should attempt to provide the best performance possible. Whether these tactics are a good idea depends on the specific application. Moreover, they are readily available to any database system.

3.3. Propositions that Result from the Necessity of an Open System

So far we have been discussing the characteristics of third generation DBMSs. We now turn to the Application Programming Interface (API) through which a user program will communicate with the DBMS. Our first proposition states the obvious.

PROPOSITION 3.1: Third generation DBMSs must be accessible from multiple HLLs.

Some system designers claim that a DBMS should be tightly connected to a particular programming language. For example, they suggest that a function should yield the same result if it is executed in user space on transient data or inside the DBMS on persistent data. The only way this can happen is for the execution model of the DBMS to be identical to that of the specific programming language. We believe that this approach is wrong.

First, there is no agreement on a single HLL. Applications will be coded in a variety of HLLs, and we see no programming language Esperanto on the horizon. Consequently, applications will be written in a variety of programming languages, and a multi-lingual DBMS results.

However, an open DBMS must be multi-lingual for another reason. It must allow access from a variety of externally written application subsystems, e.g. Lotus 1-2-3. Such subsystems will be coded in a variety of programming languages, again requiring multi-lingual DBMS support.

As a result, a third generation DBMS will be accessed by programs written in a variety of languages. This leads to the inevitable conclusion that the type system of the HLL will not necessarily match the type system of the DBMS. Therefore, we are led to our next proposition.

PROPOSITION 3.2: Persistent X for a variety of Xs is a good idea. They will all be supported on top of a single DBMS by compiler extensions and a (more or less) complex run time system.

Second generation systems were interfaced to programming languages using a preprocessor partly because early DBMS developers did not have the cooperation of compiler developers. Moreover, there are certain advantages to keeping some independence between the DBMS language and the programming language, for example the programming language and DBMS can be independently enhanced and tested. However, the resulting interfaces were not very friendly and were characterized as early as 1977 as "like glueing an apple on a pancake". Also, vendors have tended to concentrate on elegant interfaces between their 4GLs and database services. Obviously it is possible to provide the same level of elegance for general purpose programming languages.

First, it is crucial to have a closer match between the type systems, which will be facilitated by Proposition 1.1. This is the main problem with current SQL embeddings, not the aesthetics of the SQL syntax. Second, it would then be nice to allow any variable in a user's program to be optionally persistent. In this case, the value of any persistent variable is remembered even after the program terminates. There has been considerable recent interest in such interfaces [LISK82, BUNE86].

In order to perform well, persistent X must maintain a cache of data elements and records in the program's address space, and then carefully manage the contents of this cache using some replacement algorithm. Consider a user who declares a persistent data element and then increments it 100 times. With a user space cache, these updates will require small numbers of microseconds. Otherwise, 100 calls across a

protected boundary to the DMS will be required, and each one will require milliseconds. Hence, a user space cache will result in a performance improvement of 100 - 1000 for programs with high locality of reference to persistent data. The run time system for persistent X must therefore inspect the cache to see if any persistent element is present and fetch it into the cache if not. Moreover, the run time system must also simulate any types present in X that are not present in the DBMS.

As we noted earlier, functions should be coded by including calls to the DBMS expressed in the query language. Hence, persistent X also requires some way to express queries. Such queries can be expressed in a notation appropriate to the HLL in question, as illustrated for C++ by ODE [AGRA89]. The run-time system for the HLL must accept and process such queries and deliver the results back to the program.

Such a run time system will be more (or less) difficult to build depending on the HLL in question, how much simulation of types is required, and how far the query language available in the HLL deviates from the one available in the DBMS. A suitable run-time system can interface many HLLs to a DBMS. One of us has successfully built persistent CLOS on top of POSTGRES using this approach [ROWE90].

In summary, there will be a variety of persistent X's designed. Each requires compiler modifications unique to the language and a run time system particular to the HLL. All of these run time systems will connect to a common DBMS. The obvious question is "How should queries be expressed?" to this common DBMS. This leads to the next proposition.

PROPOSITION 3.3: For better or worse, SQL is intergalactic dataspeak.

SQL is the universal way of expressing queries today. The early commercial OODB's did not recognize this fact, and had to retrofit an SQL query system into their product. Unfortunately, some products did not manage to survive until they completed the job. Although SQL has a variety of well known minor problems [DATE84], it is necessary for commercial viability. Any OODB which desires to make an impact in the marketplace is likely to find that customers vote with their dollars for SQL. Moreover, SQL is a reasonable candidate for the new functions suggested in this paper, and prototype syntax for several of the capabilities has been explored in [BEEC88, ANSI89]. Of course, additional query languages may be appropriate for specific applications or HLLs.

Our last proposition concerns the architecture which should be followed when the application program is on one machine interfaced to a DBMS on a second server machine. Since DBMS commands will be coded in some extended version of SQL, it is certainly possible to transmit SQL queries and receive the resulting records and/or completion messages. Moreover, a consortium of tool and DBMS vendors, the SQL Access Group, is actively working to define and prototype an SQL remote data access facility. Such a facility will allow convenient interoperability between SQL tools and SQL DBMSs. Alternately, it is possible to communicate between client and server at some lower level interface.

Our last proposition discusses this matter.

PROPOSITON 3.4: Queries and their resulting answers should be the lowest level of communication between a client and a server.

In an environment where a user has a dedicated workstation and is interacting with data at a remote server, there is a question concerning the protocol between the workstation and the server. OODB enthusiasts are debating whether requests should be for single records, single pages or some other mechanism. Our view is very simple: expressions in the query language should be the lowest level unit of communication. Of course, if a collection of queries can be packaged into a function, then the user can use a remote procedure call to cause function execution on the server. This feature is desirable because it will result in less than one message per query.

If a lower level specification is used, such as page or record transfers, then the protocol is fundamentally more difficult to specify because of the increased amount of state, and machine dependencies may creep in. Moreover, any interface at a lower level than that of SQL will be much less efficient as noted in [HAGM86, TAND88]. Therefore, remote procedure calls and SQL queries provide an appropriate level of interface technology.

4. SUMMARY

There are many points upon which we agree with OODB enthusiasts and with [ATKI89]. They include the benefits of a rich type system, functions, inheritance and encapsulation. However, there are many areas where we are in strong disagreement. First, we see [ATKI89] as too narrowly focused on object management issues. By contrast, we address the much larger issue of providing solutions that support data, rule and object management with a complete toolkit, including integration of the DBMS and its query language into a mult-lingual environment. As such, we see the non-SQL, single language systems proposed by many OODB enthusiasts as appealing to a fairly narrow market.

Second, we feel that DBMS access should onlu occur through a query language, and nearly 20 years of history convinces us that this is correct. Physical navigation by a user program and within functions should be avoided. Third, the use of automatic collections whenever possible should be encouraged, as they offer many advantages over explicitly maintained collections. Fourth, persistence may well be added to a variety of programming languages. Because there is no programming language Esperanto, this should be accomplished by changing the compiler and writing a language-specific run-time system to interface to a single DBMS. Therefore, persistent programming languages have little to do with the data model. Fifth, unique identifiers should be either user-defined or system-defined, in contrast to one of the tenets in [ATKI89].

However, perhaps the most important disagreement we have with much of the OODB community is that we see a natural evolution from current relational DBMSs to ones with the capabilities discussed in this paper. Systems from aggressive relational vendors are faithful to Tenets 1, 2 and 3 and have good support for propositions 1.3, 1.4, 1.5, 2.1, 2.3, 2.4, 3.1, 3.3 and 3.4. To become true third generation systems they must add inheritance, additional type constructors, and implement persistent programming languages. There have been prototype systems which point the way to inclusion of these capabilities.

On the other hand, current systems that claim to be object-oriented generally are not faithful to any of our tenets and support propositions 1.1 (partly), 1.2, 1.3 and 3.2. To become true third generation systems, they must add a query language and query optimizer, a rules system, SQL client/server support, support for views, and persistent programming languages. In addition, they must undo any hard coded requirement for UIDs and discourage navigation. Moreover, they must build 4th generation languages, support distributed databases, and tune their systems to perform efficient data management.

Of course, there are significant research and development challenges to be overcome in satisfying these propositions. The design of a persistent programming language for a variety of existing HLLs presents a unique challenge. The inclusion in such languages of pleasing query language constructs is a further challenge. Moreover, both logical and physical database design are considered challenging for current relational systems, and they will get much more difficult for systems with richer type systems and rules. Database design methodologies and tools will be required to assist users in this area. Optimization of the execution of rules poses a significant challenge. In addition, tools to allow users to visualize and debug rule-oriented applications are crucial to the success of this technology. We encourage the research community to take on these issues.

REFERENCES

[AGRA89] Agrawal, R. and Gehani, G., "ODE: The Language and the Data Model," Proc. 1989 ACM-SIGMOD Conference on Mangement of Data, Portland, Ore. June 1989.

[ANON85] Anon et. al., "A Measure of Transaction Processing Power," Datamation, 1985.

[ANSI89] ANSI-ISO Committee, "Working Draft, Database Languages SQL2 and SQL3," July 1989.

[ATKI89] Atkinson, M. et. al., "The Object-Oriented Database System Manifesto," ALTAIR Technical Report No. 30-89, GIP ALTAIR, LeChesnay, France, Sept. 1989, also in Deductive and Object-oriented Databases, Elsevere Science Publishers, Amsterdam, Netherlands, 1990.

[BACH73] Bachman, C., "The Programmer as Navigator," CACM, November 1973.

[BEEC88] Beech, D., "A Foundation for Evolution from Relational to Object Databases," Proc. Conference on Extending Database Technology, Venice, Italy, April 1988.

[BERN90] Bernstein, P. et. al., "Implementing Recoverable Requests Using Queues", Proc. ACM SIGMOD Conference on Management of Data, Atlantic City, N.J., May 1990.

[BUNE86] Buneman, P. and Atkinson, M., "Inheritance and Persistence in Programming Languages," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[CARE88] Carey, M., et. al., "A Data Model and Query Language for EXODUS," Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, Ill., June 1988.

[CARE90] Carey, M., et al, "An Incremental Join Attachment for Starburst," (in preparation).

[CHAN89] Chang, E. and Katz, R., "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-oriented DBMS," Proc. 1989 ACM-SIGMOD Conference on Management of Data, Portland, Ore., June 1989.

[CODA71] CODASYL Data Base Task Group Report, April 1971.

[CODD74] Codd, E. and Date, C., "Interactive Support for Non-Programmers: The Relational and Network Approaches," Proc. 1974 ACM-SIGMOD Debate, Ann Arbor, Mich., May 1974.

[COPE84] Copeland, G. and Maier, D., "Making Smalltalk a Database System," Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.

[DADA86] Dadam, P. et al., "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View of Flat Tables and Hierarchies," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, DC, 1986.

[DATE84] Date, C., "A Critique of the SQL Database Language," ACM SIGMOD Record 14(3), November 1984.

[DATE86] Date, C., "An Introduction to Database Systems," Addison-Wesley, Reading, Mass., 1986.

[DEWI90] Dewitt, D. et. al., "A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems," ALTAIR Technical Report 42-90, Le Chesnay, France, January 1990.

[HAGM86] Hagmann, R. and Ferrari, D., "Performance Analysis of Several Back-End Database Architectures," ACM-TODS, March 1986.

[KIM90] Kim, W., "Research Directions in Object-oriented Databases," MCC Technical report ACT-OODS-013-90, MCC, Austin, Tx., January 1990.

[LISK82] Liskov, B. and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust Distributed Programs," Proc. 9th Symposium on the Principles of Programming Languages, January 1982.

[OSBO86] Osborne, S. and Heaven, T., "The Design of a Relational System with Abstract Data Types as Domains," ACM TODS, Sept. 1986.

[ROWE79] Rowe, L. and Shoens, K., "Data Abstraction, Views and Updates in RIGEL," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., May 1979.

[ROWE90] Rowe, Lawrence, "The Design of PICASSO," (in preparation).

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference on Management of Data, San Jose, May 1975.

[STON83] Stonebraker, M., "Document Processing in a Relational Database System," ACM TOOIS, April 1983.

[STON86] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," Proc. Second International Conference on Data Base Engineering, Los Angeles, Ca., Feb. 1986.

[STON90] Stonebraker, M., et. al., "The Implementation of POSTGRES," IEEE Transactions on Knowledge and Data Engineering, March 1990.

[STON90B] Stonebraker, M. et. al., "On Rules, Procedures, Caching and Views in Data Base Systems," Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., May 1990.

[TAND88] Tandem Performance Group, "A Benchmark of NonStop SQL on the Debit Credit Transaction," Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, Ill., June 1988.

[ZANI83] Zaniolo, C., "The Database Language GEM," Proc. 1983 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., May 1983.

[ZDON90] Zdonik, S. and Maier, D., "Fundamentals of Object-oriented Databases," in Readings in Object-oriented Database Systems, Morgan-Kaufman, San mateo, Ca., 1990.

ANSI OODBTG Workshop
Position Paper
Objectivity, Inc.
May 22, 1990

by
Leon Guzenda
Andrew E. Wade, drew@objy.com

ABSTRACT

Objectivity, Inc., markets an engineering database management system that is an OODBMS. We are committed to the idea that standards are beneficial for all players in the industry. We have played active roles in several standardization efforts, including the CAD Framework Initiative (CFI). This proposal discusses the motivation for a standardization effort, suggests a direction for its development, discusses cooperation with other such efforts, and offers some concrete starting suggestions.

CONTENTS

1. What to Stardardize; What not to Standardize

2. How to Approach Standardization

   2.1 Areas for Standards
   2.2 Organizations for Standards

3. Philosophy

4. Object Model

   4.1 Object Identity
   4.2 Complex Objects, Associations, Composite Objects
   4.3 The State of an Object

5. DDL

6. Transaction Model

7. DML

8. The Physical Storage Model

9. Data Exchange

10. Processing Queries

11. Compliance with the standard

12. Summary

1. What to Standardize; What not to Standardize

Objectivity markets an engineering database management system that is an OODBMS. We firmly believe that our customers want increased standardization, and that it is in the best interest of vendors to encourage and support such standards. To this end, we have participated in many standards development organizations, including: the CAD Framework Initiative (CFI), where we are charter members and chair the Storage Management Working Group, OSF, X/Open, ANSI OODBTG, and OMG.

What is the motivation for standards? In a word, *interoperability*. End users would like to mix and match applications, integrate multiple applications on the same databases, and communicate from one database to another. Application developers would like to have some ability to move from one DBMS vendor to another, at least without completely restarting. With such interoperability, DBMS vendors see the potential for more rapid market growth and can focus on each delivering their best efforts in the underlying DBMSs supporting the standards.

What should be standardized? To achieve this interoperability, the area to be standardized is the interface between the applications and the DBMS. This allows applications to be developed with some freedom to move from one DBMS to another. In addition, the standard should provide some means to communicate and transfer data from one DBMS to another. This allows the integration of diverse applications.

What should not be standardized? As much freedom as possible should be left to the DBMS developers to encourage on-going evolution and improvement of the technology and the products. In particular, the internals of the DBMS should not be standardized. No attempt should be made to impose internal architecture, or low-level design and implementation details, or data structures used internally or on disk, or internal object identifier (OID) structure, etc. The measuring stick applied should be, simply: what is the least that can be done to achieve interoperability?

The result of such a standardization effort should be incremental. Some basic, common ground can be chosen first, concrete agreements made based on that, and then, over time, this can be expanded to cover more and more territory as appropriate, and hence become more and more useful.

2. How to Approach Standardization

Even with the above restrictions, there are many areas to be explored for standardization. Here we discuss two aspects in approaching standardization: areas for standards and organizations for standards.

2.1. Areas for Standards

The following diagram (Fig. 1) is a simplified view of the areas for standardization. It can also serve as a road map for how to proceed.

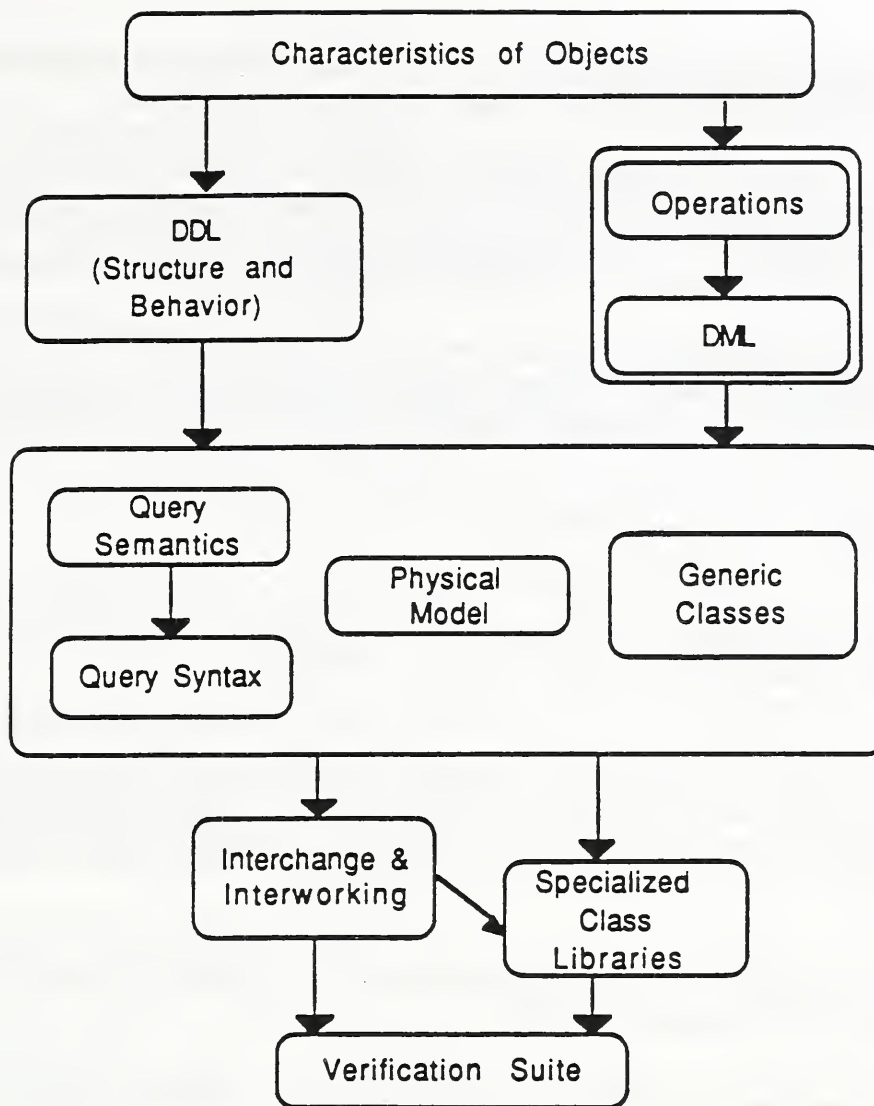


Figure 1. Areas for Standardization

The first area indicated covers the first step in an object model; viz., what is an object and what characteristics and behaviors must it support.

The next two areas are the traditional DBMS interfaces: DDL, or Data Definition Language, and DML, or Data Manipulation Language. For our purposes, we interpret these broadly. DDL, then, includes any and all mechanisms to define new object classes, both structures and methods, and by any means, whether dynamic or static, programmatic, interactive, or graphical. Similarly, we interpret DML broadly as any means to directly access, create, delete, read, write, etc., objects, including invocation of methods, lookup via indexing, triggers, etc.

The next level includes three areas: Query, Physical Model, and Generic Classes. The term "query" is sometimes used to mean interactive interface. Since we are interpreting broadly terms like DML, DDL, and Query to include programmatic as well as interactive interfaces, we use the traditional database meaning for query: a high-level associative access mechanism. Certainly the relational SQL model should be considered as a starting point. Although it may require extension, we should strive to build on what is there. The semantics should be defined, followed by syntax. The Physical Model should include

issues like clustering and DB Administration. The Generic Classes should include support for standard database functionality, including security and transactions.

Interchange and Interworking include DBMS-to-DBMS communication mechanisms as well as mechanisms to move data from one database to another (export/import). Inter-DBMS gateways might include query-level, such as the extensions to SQL discussed above, and lower-level, such as the new Remote Database Access (RDA) protocol.

Specialized Class Libraries raise the system one level higher to support various specific data models for different domains. Examples include EDIF (Electrical Design Interchange Format), PDES (Product Data Exchange Standard), the CFI Design Data Representation (DDR).

2.2. Organizations for Standards

There are many on-going efforts related to the above areas. It is important that these different groups cooperate, not only so that they can efficiently leverage each other's efforts, but also so that they converge toward one standard.

Ideally, each such group will focus on some portion of the overall effort, sharing and cooperating as needed. It remains to be seen, in some cases, exactly what the goals of the efforts are and where the overlap exists. A practical way to proceed would be for these organizations, perhaps under the auspices of ANSI, to share where each of them is headed, what area they see as most critical for their own needs, and what progress they've made to date.

Though this is not intended to be a complete list, here are some of the ongoing efforts and the areas where they are currently working:

- CAD Framework Initiative (CFI) Storage Manager (object characteristics, DML, DDL)
- CFI Design Data Representation (application data model)
- CFI Design Data Management (versioning and configuration management, etc.)
- Object Management Group (OMG) (object reference model, etc.)
- OODBTG (object reference model)
- Object-Oriented Database Manifesto (Atkinson, et. al.) (definition of OODBMS)
- OSF (OS, distributed computing facility, architecture-neutral distribution format, Motif, etc.)
- X/Open (applications environment, internationalization, endorses POSIX)
- CALS (DOD) (data interchange and sharing, PDES)
- PDES (application data model)
- ISO (OSI remote database access protocol)
- Workshop on Object-Oriented Design (WOOD) (terminology definition)
- Various ANSI efforts are related, including SQL, C, etc.

3. Philosophy

We make recommendations in these areas:

- Pragmatic, Iterative Approach
- Cooperation of Diverse Standards Efforts
- Environment to be Supported

It is amply clear that the entire area of standardization is complex. Yet, the need is strong. For this reason, and also because of the need to check evolving standards against real-world applications, we recommend a pragmatic approach: seek a useful, non-controversial subset, agree on common ground, issue this subset as a step forward towards the complete standard, and then iterate. In each step, then, vendors can support the standard, users can use it, practical feedback can be obtained, and then the work can extend to the next step.

This pragmatic, iterative approach makes more sense for some organizations than for others. Some may wish to wait for more complete results before issuing support, which is fine. This is one way in which different organizations can play different roles.

Also, different organizations can focus on different areas of standardization. For example, ANSI has an active SQL effort, so perhaps that is the place to pursue query standards. Also, ANSI may be able to play a role in coordinating the different on-going efforts.

The environment in which the OODBMS standard exists is a rich one. To be successful and widely used, it must support a wide variety of hardware platforms and operating systems in a distributed, heterogeneous network. Similarly, it must support a wide variety of languages (C++, C, Ada, Fortran, etc.) It should avoid defining new languages, but rather use existing ones. Also, it must cooperate synergistically with related applications, including Computer-Aided Software Engineering (CASE). Finally, it will best serve the industry at large if it can be supported in an incremental, perhaps layered, approach, allowing vendors to support various areas or levels of the standard as appropriate.

4. Object Model

Much has been written on the subject of reference models for OODBMSs. The ANSI OODBTG has made significant progress on their draft. OMG is in the early stages of theirs. WOOD produced results, but never published them. Perhaps most universally recognized is the work of the Object-Oriented Database System Manifesto (Atkinson, et. al., *The First International Conference on Deductive and Object-Oriented Databases Proceedings*, December 4-6, 1989, Kyoto Research Park, Kyoto, Japan). We endorse this, and support such efforts as useful in providing an intellectual framework for discourse and common terminology.

4.1 Object Identity

The concepts that must be agreed upon to begin our pragmatic, iterative process are really quite few. Principally, we must agree on the concepts of *object identity*, or unique existence of an object regardless of its state, and *object identifier (OID)*, a means by which to access such an object uniquely. We need not, and should not, attempt to standardize the internal format of this OID.

4.2 Complex Objects, Associations, Composite Objects

We propose, further, that objects may contain rich structure and methods. The methods should approach arbitrary flexibility in the languages used. Similarly, the structure should allow as much flexibility as possible. The applications interested in such systems demand that objects may contain multiple dynamically varying length components, forming *complex objects*. To support the complexity of application data structures, we need a concept of *associations*, or direct inter-object links. Groups of such associated objects can be composed to form *composite objects*, which themselves act as objects with methods (e.g., delete) operating on them.

Finally, as a special case of composite object, a *container* comprises objects (simple, complex, or composite), with each object residing within a single container. Associated objects may provide different views for the applications.

4.3 The State of an Object

An object which is currently available to a process is said to be "Open." Its inverse is "Closed." This is just one of many states. The needs of the application, the DML and the DBMS utilities should all be addressed and this adds some unusual states. For instance, "Deleted" is of no interest outside of the Storage Manager, which may have to resurrect it if the transaction fails. It is included because some part of the Reference Model or compliance tools may need to mention this state. There are also items such as Object Size which are omitted because they are probably just generic methods. Here is an initial list of states:

- Open or closed
- Transient or persistent
- Simple or complex
- Individual or composite
- Unique or versioned [linear or branching]
- Named or anonymous
- Local or remote
- Original or replica [which is checked-out]
- Vulnerable or backed up [a special type of replica until it is restored]
- Online or archived [hopefully, this is well hidden]
- Container or containee
- Protected or public [security/privacy]
- Encrypted or clear
- Compressed or expanded

5. DDL

We would prefer not to define a new Object Definition Language, although an existing standard, such as PDES/STEP EXPRESS, might be acceptable. We would like to see individual ANSI language implementations in support of the Reference Model and suggest

that C++ with some enhancements could be a good language to start with [despite its not having ANSI status at this time]. The DDL must be capable of expressing inter-object associations. We would also like to see some generic Class Definition classes in the Reference Model. This should make it easier to browse and insert definitions dynamically in a system containing DBMSs from many vendors. Ideally, the user should be able to view a definition in a form appropriate to the language to be used to access an object of that class.

Database standards have tended to ignore the issue of schema migration. The net result is that application vendors and end users have to make ad hoc decisions on how and when to introduce schema changes. If end users are to be able to maintain applications software, DBMS software, object class libraries and objects from many sources then this issue should be addressed as an integral part of the Reference Model. At a bare minimum it should cover:

- Versioning of object definitions
- Schema security [privacy], hopefully as a subset of object security
- Generic methods for upgrading object definitions and existing objects

6. Transaction Model

The standard should attempt to meet the diverse requirements of both commercial and engineering data processing. It should address:

- Atomic short and long transactions with commit and abandon capabilities
- Nested and cooperating transactions
- Transient and persistent locks
- Flexible granularities for locking and enforcing transaction semantics

7. DML

The minimum agreement on DML needed to go forward would include create/delete, open/close, and read/write. Various language bindings could be defined. We propose both C and C++. While C++ provides more power, elegance, and flexibility, C is widely used. C++ efforts might be coordinated through an ANSI committee on that language. C would be a traditional procedural interface, and could be pursued in parallel.

We suggest that at least the following are required:

- New, open, close, delete, copy, validate, upgrade, iterate, export, import, archive, online, backup, diagnose, restore, lock, unlock, encrypt, decrypt, compress, expand, name, protect and output
- Methods for connecting to multiple DBMS services and also for handling transaction semantics
- Methods for administering the physical layer of the database, e.g. Move

There should be generic classes to support the following [at minimum]:

- Object definition
- User and workgroup definition
- Security [e.g. access control lists]
- Basic versioning semantics, such as genealogy and audit trails
- Naming services [logical and physical]

8. The Physical Storage Model

CODASYL defines a physical storage model but SQL does not. We recommend defining a physical storage model using either POSIX or X/Open CAE terminology. This will not unduly constrain suppliers. It will make it much easier for the reference model to:

- Specify basic clustering capabilities
- Reconcile system namespace issues [e.g. filename]
- Provide guidance on database administration facilities.

Note that there is no constraint on the way in which objects map into this physical model. An implementor should be able to use a disk per object [extreme case!] and still be compliant. At minimum, the model should include processes, volumes and files. The OSF Distributed Computing Facility might provide a complete solution. There should be a name server for correlating logical names and their physical counterparts. OSI X.500 might be useful here.

9. Data Exchange

Defining methods in a manner which will allow users to export data, metadata, and methods to other sites without having to know the platforms and languages in use at those sites is a great challenge. The heterogeneity issues are well understood as far as the data is concerned. The language issue is more difficult. What use is an object with embedded C++ methods if the recipient only has ADA? Maybe the OSF Architecture Neutral Distribution Format will be of some use here.

The interchange of information between homogeneous DBMSs is relatively straightforward in either loosely coupled [export/import] or closely coupled [bridge or gateway] modes. The Standard should define the export/import protocol. Special attention must be paid to security issues. Close coupling should be achieved via a regular DML interface.

Exchanging data between heterogeneous DBMSs is more difficult. We believe that this Task Group should collaborate with the ISO OSI Remote Database Access Protocol Task Group to enhance their Draft Standard to cover OODBMS requirements.

A DBMS is of little use if end users find that it is awkward or impossible to build databases using software from a variety of vendors. The standard should address the following issues:

- Vendors must be able to deliver both public and proprietary object classes without fear of disclosing trade secrets. For example, the data structures needed to support a viable solid modeler were closely guarded secrets in the early stages of the development of the technology.
- Schema and class name conflicts must be easily resolvable at the end user site. An alternative would be to use a global Name Server to avoid ambiguities.
- The interfaces to standard database administration tools should be defined. This will help avoid problems in networks involving DBMSs from multiple vendors.

10. Processing queries

It may be possible to enhance SQL2 to the point where it supports queries on engineering data types. However, it will also need enhancement to cope with the semantics of single and multiple inheritence. Regardless of the outcome, we recommend that the filtering capabilities of the query language be used to qualify the initial conditions of iterators in the regular DML.

Some languages [e.g. ALGOL] failed to gain universal approval in part because they ignored the issues of formatting output. As objects will hold voice, images and text in addition to conventional data types it is important to define the standard output methods for the generic classes. These should include output in Office Document Architecture format and possibly PEX or some equivalent 3D format.

11. Compliance with the standard

The Standard may allow multiple levels of compliance, e.g. there might be a subset which is adequate for supporting conventional applications; or a C++ subset. The Task Group should communicate with interested parties within the industry to establish a suite of tests which will enable vendors to certify their products as compliant with the standard.

12. Summary

A standardized OODBMS interface protocol (or set of protocols) need not, indeed should not, dictate a DBMS's internal implementation but could greatly facilitate interoperability of applications across different vendors' platforms both today and in the future. Ideally, such a standard would address the object model, DDL, transaction model, DML, physical storage model, data exchange, and query processing. In addition, the best approach to OODBMS standards development would be pragmatic and iterative.

Objectivity is committed to cooperating in such standards efforts, and offers to participate by contributing to the effort.

The Object Standardization Challenge

William Kent

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303

1 Introduction

The challenge for object standardization, as in other areas, is how to reconcile diverse concepts of objects and object orientation. Many people can tell you in no uncertain terms what "object" and "object orientation" mean. It's made the Sunday supplements in Silicon Valley [Pollack], and there are manifestos [ABDDMS, ADBMS]. Yes, they all know, but they don't agree.

Titles that were considered for this paper include

- "The Object Paradigm" or "The Object Paradox"
- "The Object Model" or "The Object Muddle"
- "The Object Mystery"
- "The Hunt for the Real Object"

There are different object concepts rooted in the areas of user interfaces, programs, and data; some of these in turn subdivide into smaller camps. The various views seem to have much in common, though we're not quite sure exactly what; there are several efforts under way trying to figure that out [OODBTG, OMG, Kent90b]. There are also apparent differences, which we hope aren't entirely irreconcilable.

This diversity of opinion is at a very coarse-grained level, concerned with the fundamental nature of what object orientation is. Other issues, like various notions of types, classes, polymorphism, inheritance, configuration management, complex objects, multimedia data, and so on, are fine points, almost lost in the noise among the deeper questions.

The outcome seems, at the moment, that consensus will emerge on an object paradigm that expresses the essential behavior of object orientation. It is less likely that consensus will emerge on the exact meaning of the term "object" within this paradigm; it will be defined by arbitration or fiat.

2 Three Perspectives

2.1 User Interfaces

From the point of view of user interfaces, object orientation seems to mean the use of iconic graphic interfaces. "Object" might refer to the icons themselves, or to the things they represent, but the key point is that such graphic interfaces are the crux of object orientation.

The other viewpoints don't seem to insist on any particular style of user interface.

[0] ©1990 Hewlett-Packard Company

2.2 Program Objects

From the software development perspective, an object is essentially a program. Object-oriented programming emerged as a technique for constructing programs from reusable modules of code, i.e., objects. In a sense, it brings new discipline to the notion of subroutine libraries. In this view, objects are active things that communicate with each other using well-defined standardized protocols. Functionality is modularized, so that (a) common functions are provided in a common way, and (b) they only need to be implemented once.

For example, a program that prints documents would not include the code to format the documents. That would be implemented separately in a formatting program which could be invoked not only by the print program, but perhaps also by a graphic document browser. The printer and the browser would request the formatting service in a standard way. If a new program, such as a fax transmitter, also needed a formatting capability, it could call on the existing program without having to re-implement the service. The formatting program need only be written once, cutting development costs. Future maintenance is also more efficient and economical, since repairs and enhancements only need be made to one program rather than in all the programs that need to have documents formatted.

The formatting program has a certain interface by which it may be invoked, certain arguments which it will accept, and certain results that will be returned. Encapsulation, the discipline of modularization, dictates that formatting be done only in observance of these conventions. No external user or program may be aware of or depend on the internal mechanisms of the formatter. Internal implementation of the formatter may thus be freely altered without impacting any users, so long as the conventions are preserved.

Such program objects are the building blocks of reusable code from which new applications are constructed.

2.3 Data Objects

From a data point of view, objects are the things such programs deal with, e.g., documents or printers.

Only certain services can be requested on certain objects. Forbidding certain operations protects the encapsulation of objects by disallowing access to the detailed formats of their implementation. For example, if some documents are implemented by storing them under a relational storage manager, external users are not allowed to operate on the underlying tables and tuples; the configurations might change in different implementations.

Data objects can be grouped into "classes" (or "types") of things on which the same services can be requested. For example, documents are characterized as the class of things which can be edited, formatted, and printed.

Data objects are created by end users (directly via user interfaces or indirectly via application programs), and there tend to be many of them. They have distinct identity; each one created is distinct from all others.

Data objects are characterized as being transient or persistent. When the execution of a program finishes, the transient objects it created disappear, while the persistent ones persist.

2.4 They're the Same Thing

There is a viewpoint which tries to unify the notions of program object and data object, saying they are the same thing. In effect, each document is considered to "contain" its own editor, formatter, and printer. Such objects are spoken of in an active way. One asks a document to print itself; if the document contains diagram objects, then the document asks the diagrams to print themselves

during this process.

This metaphor does provide a convenient explanation of "polymorphism", or "generic operations". In spite of the desire to isolate the printing or formatting capability into a single program each, it often happens that different kinds of documents need different kinds of printing or formatting programs. Different implementations may be needed for documents with simple text, documents with multiple fonts, documents with graphics, and documents with color. Or there may be different implementations for documents produced by different word processors. Generic operations shield external users from these distinctions, by requiring that the printing and formatting services be requested for all documents in a uniform way. A user simply requests that a document be printed or formatted, without concern for the alternative implementations. The appropriate programs (also called methods) will be invoked for each document.

A common way to explain this is to say that each document contains its own printing and formatting methods. A request to print or format a particular document invokes its own printing or formatting method.

However, it is common knowledge that object systems are rarely implemented in this way. Classes of objects are further partitioned into subclasses according to their different implementations. Each edit, format, and print program is typically kept in one place, associated with the corresponding object class. Space isn't allocated for a copy of these programs each time a new document is created.

The utility of the unity metaphor as an explanatory device is uncertain. It's not likely that many people will really believe that each memo they write contains a copy of WordStar, or that each spreadsheet contains a copy of Lotus-1-2-3.

The unity of the one and the many seems a mystery, difficult to reconcile with other characteristics of program and data objects.

Program objects are typically created by system or application developers; data objects by end users.

Software development methodology urges singularity of programs: if at all possible, there should only be one print program and one formatting program. In contrast, there naturally are multiple occurrences of data objects.

To the extent that a program object has an identity, it is distinct from any of the data objects it can operate on. The print program has a different identity from any of the documents it can print.

In the programming world, transience is taken to be the default for data objects, but not for program objects: they tend to survive their execution (deadly pun).

3 Who Gets the Message?

A request to print a document is portrayed in many object models as a message to an object. Which object? If we are thinking in terms of program objects, it must be to the print program. But that's not consistent with generic operations. The message must presumably go to the document, in order to pick the right print method for that document.

The message metaphor, closely linked to the unity metaphor, is useful up to a point. But if we let ourselves be aware that data objects rarely contain their own individual copies of programs, then we know that what "really" happens is that the appropriate print method is chosen on the basis of the class to which the document belongs. By whom? Well, there is a "system" in the background taking care of such things as parsing messages, sending them to the right places, returning results, and so on. We can just as easily say that the system fields the message, choosing the appropriate method based on the class of the document.

The message metaphor is difficult to sustain if we don't just ask to print a certain document, but

ask that it be printed on a certain printer. To whom is the message addressed? Are we asking the document to print itself on the specified printer, or the printer to print a specified document? Why should it matter?

The unity metaphor is similarly strained. Suppose there are different kinds of printers, in addition to different kinds of documents. We might have different methods for printing different kinds of documents on different kinds of printers, and the choice of method depends on both the document and the printer. Should we still say that the method is "in" the object? In the document or in the printer? Why?

An alternative to the messaging metaphor is a general operator paradigm, in which a service is requested as an operation with one or more operands. The appropriate method is chosen by the system, based on the classes of the operands. A method may be jointly owned by one or more classes, corresponding to its operands. The messaging paradigm can be treated as a special case, in which the "recipient" is considered to be the first (or only) operand.

4 Data Objects: Structure vs. Behavior

Where is data in all of this?

There are two approaches to data objects: structural and behavioral.

The structural approach continues the data modeling tradition of describing data in spatial terms, using visualizable formats such as tables or hierarchies. Structural object orientation is largely characterized by more complex structures than prior models, usually defined as recursive applications of constructors for such things as sets, lists, and tuples (or records). The goal is to provide direct support for the more complex structural relationships observed in real-life objects.

In the behavioral view, information is defined entirely in terms of the results of operations performed on objects. Documents are characterized, for example, by the fact that it is possible to ask to know who is the author of a document. Another operation which changes the author of a document is described as altering the results of the author request, rather than in terms of stored data formats. The content or state of objects can be characterized as the results returned by a certain set of operations.

The complex composition of objects can also be described behaviorally, in terms of propagated operations. The fact that a certain document contains a certain diagram is evident because a request to display or destroy the document automatically displays or destroys the diagram. It is not described in terms of a spatial layout such that the space occupied by the diagram is a subset of the space occupied by the document.

In a sense, the real difference between the two approaches has to do with who defines the allowable operations. The structural approach is characterized by a fixed set of operations defined by the system implementers, for constructing and accessing the data structures. The allowable operations in the behavioral view are defined by the application developers, with semantics appropriate to application-specific objects.

In the behavioral view, data structure ought to be hidden, part of the encapsulation. Data structures may be used by methods as part of their internal implementations, but not by external object users. This might provide a basis for positioning both approaches in the object paradigm: structural object orientation is usable by methods which implement operations on objects, while the behavioral approach ought to be taken by external users of objects.

5 The Object Paradigm

Is there one?

Certain elements seem to be emerging as common to all views. An object system might be characterized in terms such as those listed below. This is just an illustrative list, not intended to be complete; we are not competing with the manifestos. Also, we deliberately avoid the word "object".

- Subjects are things which may occur as operands or results of requests for service.
- The set of services which may be requested, and the subjects on which they may be requested, is controlled (encapsulation).
- The set of services and subjects is extensible, in terms semantically meaningful to the application domain.
- The implementation of services is designed in modular fashion, maximizing reuse of code.
- The methods which implement services are typically able to request different services on different objects, corresponding to internal implementations.
- A requested service might be provided by different methods, depending on the operands of the particular request. These alternative implementations are invoked in a uniform manner.
- Genericity: programs don't call each other directly. The method that responds to a request depends on the nature of the operands, beyond the control of the requestor.
- There are standardized syntaxes for requesting services. This may include graphic interface conventions.
- Certain standardized classes and services are available.

The term "object" might turn out to mean a subject, or a requestor, or a method, or some combination, or something else.

Database might evolve along several lines in this context. The structural view might be exposed to end users, providing richer sets of system operations for manipulating data structures.

In a strictly behavioral approach, database will present two interfaces [Kent90a]. Object users will see a purely operational interface, not manipulating data structures directly. Structures supported by the database system will be exposed only to the methods which implement object services.

From the object users' perspective, there is no object data model, just an object model. Database would be perceived externally as providing services such as persistent storage, recovery, concurrency control, and so on, but not specific storage structures.

Program objects and data objects are unified, not by becoming the same thing, but by being managed uniformly in an integrated facility.

6 Conclusions

Standards bodies are currently exploring major questions about the context before plunging into the specifics of object database standards.

Consensus needs to be established on the overall object paradigm, and the meaning of "object" and related terms. The relationship between the structural and behavioral approaches needs to be clarified, together with the interfaces implied by each.

Only after such context has been established will it be meaningful to grapple with specific issues of object model details, interfaces, syntaxes, and semantics.

References

[ABDDMS] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik. The Object-Oriented Database System Manifesto. Proc First Intl Conference on Deductive and Object-Oriented Databases, Dec., 1989, Kyoto, Japan.

[ADBMS] The Committee for Advanced DBMS Function. Third-Generation Data Base System Manifesto. Memorandum No. UCB/ERL M90/28, UC Berkeley, April 1990.

[Kent90a] William Kent. The Evolving Role of Database in Object Systems. HPL-90-04, Hewlett-Packard Laboratories, Feb. 1990.

[Kent90b] William Kent. A Framework for Object Concepts. HPL-90-30, Hewlett-Packard Laboratories, April 1990.

[OODBTG] Technical Report of the ANSI/X3/SPARC/DBSSG Object-Oriented Database Task Group. (in preparation).

[OMG] Object Management Group Standards Manual. (in preparation).

[Pollack] Andrew Pollack. 'Industrial revolution' retools software. *San Jose Mercury News*, May 6, 1990, p. 1F.

Application Object Model
for Engineering Information Systems[1]

Jonathan W. Krueger
Honeywell Systems and Research Center
Minneapolis, MN

Abstract

An object model for application development is introduced in the context of Engineering Information Systems (EIS). The model shares features with several popular models and offers some less common ideas, such as operation-dependent state closure. The model is also unique in its position within the environment: it acts as a portability view independent of the underlying storage servers. The view-mapping is made most efficient however, if the object model of the underlying servers matches the structure of the EIS Application Object Model (AOM); OODB standardization is significant to the AOM in this respect.

Background

The Engineering Information System (EIS) program [3] was created by the government in response to increasing concern about engineering information exchange between:

- Contractors and the government;
- Contractors on a single team;
- Contractors not on the same team but whom the government wishes to encourage to share and leverage technical data; and finally,
- Tools at a single site.

[1] This work is supported by the Air Force Wright Research and Development Center's EIS project under contract F33615-87-C-1401.

This information exchange should be enabled even when the two endpoints were not originally designed to facilitate this exchange.

The EIS program addresses heterogeneity of hardware and software platforms, data formats, tools, site-specific policies and methodologies, and interfaces. The program's goal is to produce a consolidated approach to a broad set of functional and other requirements. This approach consists of proposed standards and guidelines for services which, if used, enable and accelerate a trend toward uniform engineering environments and information exchange.

A given EIS site might use proprietary and/or commercial toolsets. Site tailoring includes the specific design policies which the EIS is to enforce for an organization. A tool can be a collection of functions operating in concert with a user interface, or it can be a single function providing a specialized service.

From the EIS perspective, database management systems and file systems — collectively called data servers — are a special kind of tool whose function is to administer persistent data. There is an additional facet to data servers in that many contemporary tools have already achieved the separation of data from computation through the use of database management systems. The EIS is intended to support attachment of these data and file servers while maintaining the role of intermediary between tools and data.

EIS Architecture

EIS stands on two legs — the Engineering Information Model (EIM) and the framework.

- The Engineering Information Model (EIM) is a semantic model of the information flowing through the framework. The EIM has a candidate standard derivation called the Reference Schema. The Reference Schema is a set of ob-

ject types which form a specific, logical organization of data and operations that the framework can manipulate. The EIM and the Reference Schema may be extended with site-specific semantic models and types, respectively, to capture the meaning and structure of data particular to that site.

- An EIS Framework contains the automated services, embodied in software, which support the use of the EIM to manage and control the data and activities of the engineering process.

The EIM is a conceptual model of administrative and electronic design information (see Figure 1). It records common concepts and makes their meaning explicit for the primary purpose of aiding effective communication. The EIM is important for everyone involved with an EIS. A CAD tool builder uses the EIM to find common names and definitions for the information he expects to process. A framework vendor uses the EIM to identify common operations and consider whether to build them into his framework. An administrator of a CAD environment uses the EIM to quickly understand the scope and organization of EIS information. In this way he can plan adaptations of his current system as he begins to implement EIS concepts.

The EIS Framework is a collection of software services used by applications to store, retrieve and manipulate the modeled by the EIM. The various services can be organized roughly into layers as shown in Figure 2.

Starting at the bottom of Figure 2, delivery systems are computing platforms bundled with operating system software, communication hardware and peripherals. The portability services fill in the services that some delivery systems do not address, such as bitmap graphics and higher-level network communications (above the ISO/OSI Transport level). Newer operating systems are incorporating these services. The "servers" layer is populated mostly by database management systems, data format filters, and special-purpose computing resources (such as a simulation accelerator).

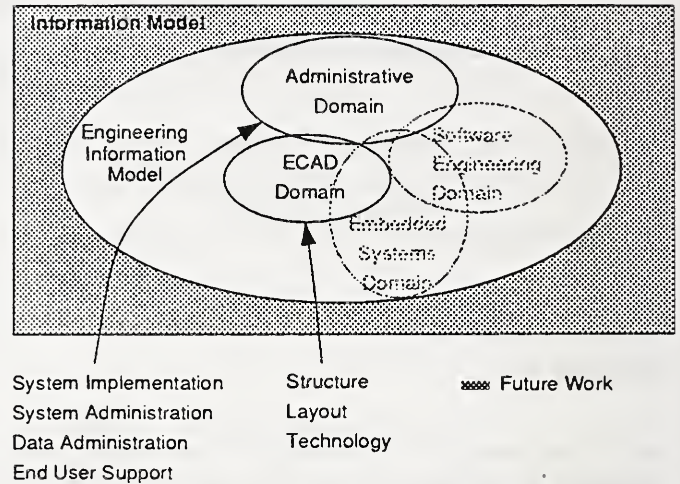


Figure 1: Domains of the Engineering Information Model

The Object Management System (OMS), is responsible for providing access to objects and operations. In a sense, it is the common "bus" to which everything connects, including tools and data servers. Data servers are files, file systems, databases, database management systems, computer memory, and anything else that stores data. By being on the same bus (sometimes via adapters) tools and their users more readily gain controlled access to data in the various servers.

The Application Object Model (AOM) is a layer built on top of the OMS to simplify the development of applications. It wraps and projects the OMS services to yield a view that resembles several popular object systems. The AOM is the primary focus for this position paper because it has the most bearing on the standardization of object-oriented database management systems (OODBMS). EIS framework performance goes up as the AOM and underlying OODBMSs look more like each other, because the OMS has to do less view-mapping.

Environment Services encompass a broad range of services of interest to application developers or end users. Topics include configuration management,

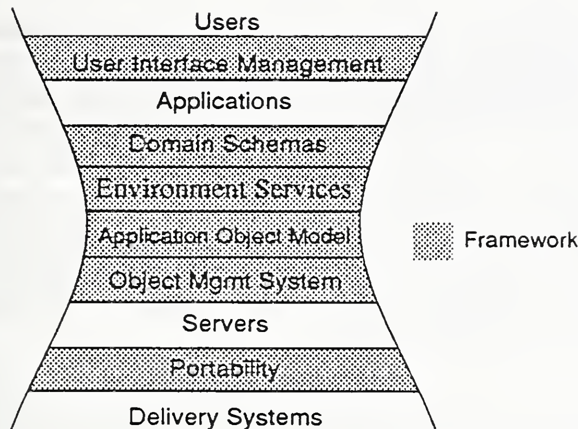


Figure 2: EIS Model and Framework

inter-EIS information exchange, policy management, audit trails, etc. Policies are used to automatically determine when system activity is needed and initiate it (system administrators can inject their own policies to tune system behavior to specific needs).

User Interface Management services couple users with programs. This involves visually organizing and portraying data, as well as interpreting and dispatching keyboard and pointer events to the appropriate programs.

The remainder of this paper will elaborate the Application Object Model.

Application Object Model

The EIS Application Object Model (AOM) is a fairly typical model for describing and manipulating objects. The AOM provides a fixed, high-level view that supports tool portability. The AOM intentionally shares many features with existing object models such as Smalltalk-80[2] [2], the Common Lisp Object System (CLOS) [1] and C++ [5].

As in nearly all object-oriented systems, AOM objects have three essential characteristics: they exhibit *behavior* when stimulated and they have *state*; objects also have identity, that is, they can be referred to in a way that is independent of their behavior and state.

Object Behavior

Behavior is elicited from an object by applying *operations* to it. An operation may take a set of parameters, with each parameter being an object. The object given as the first parameter in an operation request is called the *receiver*.

There are usually one or more *functions*[3] specified for the receiver's type that the system may choose from to serve the operation request. The function(s) chosen depends on the types of the parameters. Once a function is chosen, it is given control and may do just about anything, including manipulating the state of the parameter objects, and applying other operations.

There are two kinds of functions that play a role in servicing an operation request, namely *primary functions* and *encapsulation functions*. A primary function constitutes the main body of an operation's behavior. Encapsulation functions come in two types — *before* functions and *after* functions — and, as their name implies, encapsulate a primary function or inherited operation by executing just before it or just after it.

Operations and Functions

A *function* identifies the body of code that elicits specific behavior on an object. An *operation* is an interface to a set of one or more functions. There are two kinds of operations: type operations and instance operations. The classification is primarily conceptual, since the machinery to carry them out is the same for both. Type operations are those that can be applied to a type object. An example of such an operation is create(typeObject). The receiver of this operation is a type object, and it creates an instance of that type.

[2] Smalltalk-80 is a trademark of Xerox Corporation.

[3] Also known as methods

Instance operations can be applied only to instances of the type for which they are defined. An instance operation on the hypothetical type Pen might be drawLine(instanceOfPen, x1, y1, x2, y2), causing a line to be drawn from (x1, y1) to (x2, y2), stylized according to the pen's characteristics.

As with types, functions and operations are both modeled as objects. The essential components of a function object are its name, signature and set of implementations. The signature identifies the input and return parameter types for which it is defined, A function implementation is a body of code that implements the behavior.

An operation is a name that is shared by one or more functions, and contains three ordered sets, containing the before, primary and after functions with that name. The system uses the operation name, together with a set of actual parameters and their respective types, to select the functions to carry out the operation. The selection process is called operation resolution, described later.

Encapsulation Functions Encapsulation functions support the tailoring of inherited operations. In all respects they are regular functions, and therefore have the same characteristics as any other function. Since they are not operations, they themselves do not have before or after functions.

Before functions and after functions can be used to augment the behavior of an inherited operation to better suit a new type. For example, suppose the following two types exist:

- **ElectronicDesign,** which defines the function simulate
- **IntegratedCircuit,** which inherits the function simulate

and ElectronicDesign is a supertype of IntegratedCircuit.

Suppose that IntegratedCircuit requires a slightly modified implementation of simulate, where a clock is started to record the simulation time. Instead of overriding the inherited function, the IntegratedCircuit type could define a *before* function and an *after* function for simulate to start and stop the clock, respectively. Whenever simulate is called, its before function is first executed, then simulate is executed, and finally its after function would be executed. This is illustrated in Figure 3. Encapsulation functions are not mandatory, but they will be executed if they exist for a particular function.

Encapsulation functions are also useful for implementing integrity constraints on instance variables, such as enforcing the ordinality of a relationship (e.g., one-to-many). This can be done by registering the constraint-checking code as encapsulation functions on the instance variable accessor operations.

Encapsulation functions can nest through inheritance. If an operation is inherited, the supertype's before, primary and after functions are combined into a unit for the subtype, which views it as a primary function. The additional encapsulation functions may be defined for the subtype which are applied before and after the inherited operation. For example, suppose that encapsulation functions are defined for the ElectronicDesign type's simulate function. The IntegratedCircuit subtype inherits simulate and its associated encapsulation functions as a unit. New encapsulation functions are placed around the inherited unit, as shown in Figure 3. Similarly, a subtype of IntegratedCircuit could define an additional layer of encapsulation on simulate.

In response to a client's request for the simulate operation, the system:

- Executes all before functions defined for simulate. The function defined by the IntegratedCircuit type is executed first; the function defined by the ElectronicDesign type is executed second.
- Executes simulate.
- Executes all after functions defined for simulate. The function defined by the ElectronicDesign

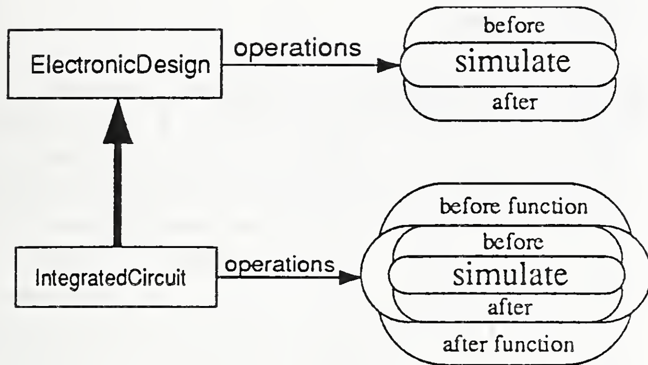


Figure 3: The *simulate* operation as seen by the IntegratedCircuit type

type is executed first, followed by the function defined by the IntegratedCircuit type.

Operation Resolution Operation resolution is the process used to find the function(s) needed to satisfy an operation request. When only one function is defined for a particular operation, the resolution process is trivial. However, when more than one function exists the system must resolve which function(s) to apply based on the objects passed as arguments. For example, consider two types, SoftwareComponent and HardwareComponent, with the following operations:

SoftwareComponent :
edit(instance) – invokes an editor
compile(instance) – invokes a compiler

HardwareComponent :
edit(instance) – invokes an editor
displayOn(instance,format) – graphically displays the instance
simulate(instance,stimuli) – runs a simulator on the instance with the given stimuli

In this example, the operation compile can be executed on an instance of SoftwareComponent but not on an instance of HardwareComponent. Similarly, *displayOn* and *simulate* are specific to HardwareComponent. However, the edit operation can be requested for instances of either type.

The user need not specify which function to apply; the system resolves the operation request to the appropriate functions based on the parameters given in the request. If an instance of SoftwareComponent is passed as a parameter to edit, the system will select the function defined for the SoftwareComponent type; similarly, if an instance of HardwareComponent is passed, the function defined for HardwareComponent will be selected. If an instance of a different type, say HostComponent, which has no edit function, the system will reject the request and raise an exception.

The edit operation described above accepts a single parameter. To see how multiple parameters are used in resolution, consider three different functions for HardwareComponent displayOn: one displays an instance on a Sun workstation, one displays an instance on an Apollo workstation, and one displays an instance on a laser printer. Each function takes an instance of HardwareComponent, as follows:

- displayOn(instanceOfHardwareComponent,sunFormat) – displays an instance on a Sun workstation.
- displayOn(instanceOfHardwareComponent,apolloFormat) – displays an instance on an Apollo workstation.
- displayOn(instanceOfHardwareComponent,laserFormat) – displays an instance on a laser printer.

Upon receiving a request for displayOn, the system first examines the receiver (first argument). In this case, the type of the receiver narrows down the set of applicable functions to the above three. Examination of the second argument carries the resolution to a single function.

The system raises the exception "functionResolutionConflict" when it cannot resolve the operation re-

quest to a single function. [4] An EIS installation might choose to handle such an exception by completing the resolution based on an ordering such as chronological by registration time. A framework vendor might choose to provide a default exception handler that could be removed or replaced according to a site's needs.

Likewise, the exception "interfaceMismatch" is raised if the resolution algorithm fails to come up with a function whose interface pattern matches the supplied parameters. One interesting way to handle this exception would be to attempt to convert the parameters to types that *do* match an available function interface.

Summary of Operations

Operations and functions are used to elicit behavior from an object. From an application standpoint, the only way to interact with any object is through its operations and functions. They are made visible to programmers in language-specific ways, for example function calls in structured languages such as Ada and C.

The requested behavior occurs when one or more functions corresponding to the operation are executed. The system chooses the function(s) to execute based on the types of *all* parameters passed in the call. Depending on the arrangement of the type hierarchy, functions registered, and encapsulation functions installed, different functions may be applied. Inheritance and encapsulation functions support the ability to build up and tailor object behavior. The AOM does not specify whether the operation → function → implementation binding is established at compile time or at run time. In a compiled environment, recompilation may be necessary to regain strict AOM semantics.

[4] A compiler could catch many of these conflicts.

Object State

The state of an object, as perceived through the AOM, is defined by the contents of its instance variables[5], as shown in Figure 4. The contents of an instance variable are retrieved and updated through accessor operations that hide the implementation details (storage format and computation). The implementations of some accessor operations may be automatically generated by the EIS framework, while others may be hand-crafted either in part or in whole.

Instance variables that hold elements of the basic types (integer, float, string, etc.) are ***attributes;*** instance variables that hold object references are ***relationships***. Attributes contribute to the ***absolute state*** of an object; relationships contribute to the object's *relative state*.

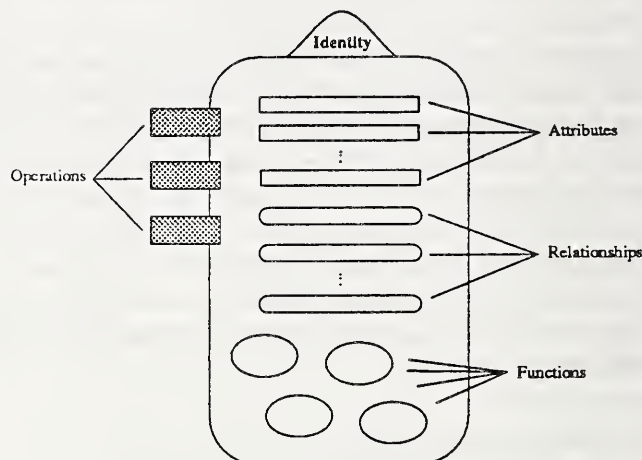


Figure 4: A Prototypical AOM Object

The distinction between attributes and relationships is useful when exchanging objects between sites or applications. An object's absolute state is independent of all other objects and can be exchanged readily simply by transferring the contents of the attributes. The relative state is, by definition, dependent on other objects. Each relationship references one or more objects that an application could regard as an essential

[5] Also known as slots.

part of the original object. To exchange the relative state of an object, some of the referenced objects may need to be part of the exchange. Since those objects have relative states as well, the algorithm for determining which objects to transfer is conceptually recursive. It may not be necessary to transfer all of the relative state however, since the applications involved may be able to ignore some of the referenced objects.

The following subsections describe the general characteristics of instance variables and specific characteristics of attributes, relationships, and inherited instance variables.

Instance Variables

An instance variable may be an attribute or a relationship. Attributes contain values and relationships refer to objects. Any instance variable may be *atomic* or *aggregated,* constrained or unconstrained.

Atomic vs. Aggregate An atomic instance variable has only a single object or reference; an aggregate instance variable can have more than one. The OMS defines several kinds of aggregation, including sets, lists, and arrays. Aggregate instance variables have additional access functions not found for atomic instance variables, e.g., add, remove and select.

Constraints Certain system-enforced constraints may be associated with all instance variables. The AOM defines standard constraints for uniqueness, type, and enumeration. Other constraints may be placed on instance variables through the use of before and after functions on the instance variable update operations.

Uniqueness The scope of uniqueness for an instance variable emcompasses all instances of the type and its subtypes (the subtypes inherit the instance variable). Applications must always specify an initial value for such an instance variable when creating an instance; instance variables defined to be unique must not have a default initial value specified, since a default would only be useful for the first instance created.

Unique instance variables are analogous to keys in a relational database and make it easier to locate a single instance through query. The OID also uniquely identifies the object. In fact, an object id is unique across all types, whereas unique instance variables are unique only within the domain of a type. However, the contents of instance variables are usually easier to work with than an OID since an instance variable typically contains more mnemonic information, such as a user-supplied name. An OID can be cryptic and seemingly arbitrary. Furthermore, the OID of an object may change as it moves from EIS to EIS[6], but the contents of its instance variables will not change unless explicitly modified.[7]

Type Constraints Instance variables may be restricted to contain only certain types of data. The system allows updates to such instance variables only if the new value (or reference) is of the specified type. The allowed types include the specified type and all its subtypes. For example, a relationship instance variable constrained to type FunctionalUnit can reference an instance of type Cell as long as Cell is a subtype of FunctionalUnit.

Enumeration An enumeration defines a domain by listing, or enumerating, particular items. An enumeration constraint prohibits instance variable updates that have items from outside the enumeration. Enumeration overrides type constraints; that is, if an instance variable has both an enumeration and a type constraint, the enumeration takes precedence. Enumerations are generally more specific than types (e.g.,

[6] But an object's OID is always the same on a given EIS.

[7] This is an oversimplification. Some instance variables, especially relationships, may change when moving to a different EIS, depending on the similarity of the schemas between the two systems.

"fred," "julie," and "terry" vs. type String), so a type constraint would have no effect on an enumerated instance variable.

Default Initial Contents When creating an instance of a type, the application may supply the initial contents for some of the instance variables; the rest of the instance variables are initialized using defaults specified in the instance variable definitions. For example, consider a type Design with a String attribute called "name" whose default initial value is "noname." An application could either create an instance of Design with a specified name, or it could accept the default "noname."

The type of an initial value must be compatible with the constraints governing the instance variable, or the system will reject the instance variable definition. Using the above example, the name instance variable could not have the Integer value 5 as an initial value. The system would raise an exception when attempting to construct the name instance variable for type Design..

Attributes

An attribute instance variable can be defined to contain either an atomic value, or an aggregate of values. Atomic values are single, elemental values commonly found in programming languages and are classified into basic types. The AOM recognizes integer, float, character,string, boolean and byte ("unsigned char" in C parlance).

Relationships

Relationships always reference other objects (with OIDs). As with attributes, relationships may refer to a single object or an aggregate of objects. Relationships may also be declared *inverse,* and may carry *property lists.*

Inverse Relationships Inverse relationship instance variables always exist in pairs. The specification of an inverse relationship must always state the other relationship instance variable acting as its inverse. For example, consider two types, Board and Chip, where Board defines the instance variable contains to be the inverse of Chip instance variable isPartOf, as illustrated in Figure 5. When an update to $Chip_1$ is made that removes $Board_1$ from its isPartOf instance variable, $Chip_1$ is automatically removed from the $Board_1$ contains instance variable. An inversion is always two-way. An update to Board automatically triggers an update to $Chip_1$ and $Chip_2$.

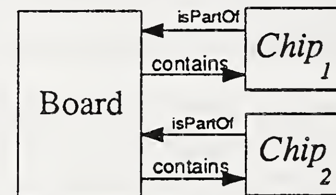


Figure 5: Example of Inverse relationships

Inheritance of Instance Variables

An inherited instance variable assumes all the characteristics of the original definition. Some characteristics of the inherited instance variable can be changed for the new type. The new definition must be *more* constrained than the inherited definition. For example, a type constraint can be narrowed to a subtype, or an element may be removed from an enumerated domain. An attempt to relax or shift constraints on inherited instance variables will be met with an exception condition; that is, it is not allowed to shift a type constraint upward in the type lattice, or to an unrelated type.

All instance variables of the supertypes are inherited; an inherited instance variable cannot be "masked out." It is possible, however, to constrain an instance variable so much that it is effectively unusable.

State Closure

Backup, archival and information exchange operations make use of relationship properties to selectively track down dependencies for the purpose of managing the state of a composite object. This subsection describes relationship properties in more detail and examines their use.

The description of an object can, and often does go beyond the contents of its attributes and relationships. An object's attributes contain values which need no further description, but its relationships contain references to other objects which may be important for understanding the original object. Likewise, the referenced object may in turn reference still more objects. Without some guide to limiting the extent of these relationships, all of the objects in the system might be considered "part of" the original object.

Relationship properties are a flexible mechanism for controlling the propagation of operations that track relationships. In particular, [4] describes two generic properties which can be used for this purpose, namely *propagate deep* and *propagate shallow*. Refinements of these properties can be used in EIS to give the type creator the flexibility to define the dependent objects for instances of its type for a particular operation performed on that object.

By defining properties associated with operations that deal with relationships (e.g. backup, archive, copy[8]), the description of an object can be tuned. Some examples of properties on relationships are: copyDeep, backupDeep, archiveDeep, changeDeep, versionDeep.[9]

A deep property associated with a relationship means to propagate the object referenced via the relationship recursively. Since the initial object to be operated on may have several relationship slots and each of relationship has a separate property list associated with it, both deep and shallow (no) propagation can occur in a single operation request.

The properties described in the previous paragraphs are used to used to determine what relationships an object is dependent on. The term ***state closure*** is used to describe the list of objects needed to describe a particular object for a requested operation. The state closure of an object can be determined in the following manner:

1. The requested object is placed in the state closure.
2. For every object placed in the state closure, a breadth first search algorithm is followed to determine new objects to be added in the state closure.
3. The relationships are examined for properties associated with the requested operation (e.g. copy, backup, archive, change, version). Whenever the relationship slot has the deep property, it means that the object referenced by the relationship is needed for the description. If it is not already in the state closure, it is added.

Object Identity

Every object has its own identifier, called an *OID* (object identifier). An object is assigned an object identifier at the time the object is created. Once established, an OID always refers to the same object.

An object *handle* is an indirect reference to an object, such as a query or a name. Suppose an application has two different handles, α and β. If α resolves to the same OID as β, then α and β are handles for the same object. Two objects are said to be equivalent if and only if their states are identical that is, if the contents of the attributes and relationships of one match those of the other. Objects that are equivalent are not necessarily identical (the same object).

Objects that share the same instance variable definitions and functions are instances of the same type.

[8] Exchanging an object with another EIS can be considered a special case of "copy."

[9] Note: <op>Shallow properties on a relationship are implied by the absence of an <op>Deep property.

An object is also an instance of the supertypes of its type. Each type is represented by a type object. The type object's instance variables and functions are applicable only to that object. The contents of some of the instance variables describe the structure and behavior (instance variables and functions) of the instances of that type. An instance is most often created by the execution of a type function, e.g., "new".

Basic types, such as integers, floats and strings, have literal representations that completely define their state and identity.

There are aggregate types, such as set and list, that contain a collection of items. Aggregates restrict their membership to a particular type of value or object. For example, an attribute could be declared to be a "SetOfInteger," or a relationship might be declared as a "ListOfChip," where Chip is a type. The contents of an aggregate object may change without affecting the identity of the aggregate itself.

New types can be defined in terms of old types by inheriting the old descriptions. In setting up a new type, the definer can instruct the system to include the definitions of one or more pre-existing types as part of the new type definition. The reused types are called *supertypes* of the new type; the new type is called a *subtype* of each supertype. Inheritance involving only one supertype is called "single inheritance." *Multiple inheritance* refers to the case where more than one supertype is used. Multiple inheritance is essential for EIS in order to support basic capabilities which can be "mixed in" through inheritance (rather than referral) with application-specific capabilities.

Type objects are instances of *metatypes*. The Application Object Model does not currently specify metatype structure or behavior. As of this writing, metatype design is still considered an art, and a hornet's nest of specification issues. The ANSI CLOS group, the furthest along in object-model standardization, has not yet developed a metatype specification even though their object-level specifications are relatively stable. One of the many reasons is that metatypes are often the focus of subtle optimization techniques that can have a significant impact on system performance.

References

[1] Bobrow, D., et al, "Common Lisp Object System Specification", ANSI working document, 1987

[2] Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983, Reading, Massachusetts

[3] "The Department of Defense Requirements for Engineering Information Systems" Institute for Defense Analyses report, July 2, 1986; Joseph L. Linn and Robert I. Winner, eds., Alexandria, VA.

[4] Rumbaugh, J., "Controlling Propagation of Operations Using Attributes on Relations," Proceedings of OOPLSA 1988.

[5] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 1986.

Principles for persistent object access

Fred Loney
Mentor Graphics Corporation

Summary

Mechanisms for implementing persistent object access are described. Criteria and opportunities for standardization are identified. Principles are suggested for the development of an object-oriented database. These principles constrain the interaction of embedded DBMS constructs within the host programming language and indicate the appropriate use of existing language features vs. extension of the host language with new features to support persistence.

1. Introduction

Object orientation offers a new development paradigm for a class of applications not well-served by traditional database technology. These applications are characterized by highly interconnected, typed objects of varying size and format. The applications typically occur in the context of a distributed processing environment. Besides the usual database amenities, an object-oriented database offers strong identity for persistent objects and seamless integration into an object-oriented programming language.

These characteristics dictate a focus for standards. OODB standards should reflect the need for encapsulation, host language integration and distributed access to hetereogenous databases. Standards are warranted when the benefits of conformance exceed the potential for competitive advantage by differentiation.* It is therefore useful to identify common aspects of OODB features that offer essential functionality,

---

* In point of fact, standards lag implementations and are more strongly correlated with market presence than intrinsic merits. The example of ANSI SQL is instructive here. The conceptual foundation of the relational data model is simplicity itself. Nevertheless, the standard wrought by the ANSI crucible abuses the model, for example by countenancing "set" manipulation that admits of duplicate members.

The true value of OODB standards at this early stage is the opportunity to nudge developers ever so slightly towards a reasonable implementation. With this in mind, we demur at the heady issues of object ontology in this paper, and address instead more pedestrian concerns of object use.

but hold forth relatively little promise for productive differences. The target of standardization is the *mechanism* presented to the client of database services.

This paper considers the mechanisms for implementing persistent object access. These include transparent use of host language constructs, special-purpose functions, database classes and programming language extensions. Section 2 discusses pointer dereference as a standard approach to object access. Section 3 discusses characteristices of object activation. Encapsulation of database objects is considered in Section 4. Conditions for extending the host programming language rather than defining database classes are developed in Section 5.

We note at the outset that when we suggest commonality of features, we are adopting the perspective of the *client* of OODB services. A common approach to the external characteristics of a feature should not be construed as advocating a common approach to the internal implementation of the feature. About this, we remain silent.

2. Object dereference

Our starting point is the semantics of a host object-oriented programming language. For better or worse, the operational semantics of the object data model is largely defined by the programming language(s) it supports. We call the standard reference model for the host programming language supported by the DBMS the *reference language* of the OODB. We posit the existence of such a language standard, and remark in passing that OODB standardization depends on language standardization. Seamless integration of the OODB with the host language motivates our first

Principle: Use the reference language to implement database features wherever possible.

In most object-oriented programming languages, object identity is established by an object-oriented pointer (OOP). Access to objects is gained by dereferencing the pointer. Persistent object access is the crux of an OODBMS data manipulation language. The principle suggests that persistent object access is best implemented by using a *persistent OOP*, a "smart pointer" which extends the behavior of the dereference operation. We thus get the following

Corollary: Persistent OOP dereference is the standard data manipulation language.

Alternative methods may be available to, for example, recursively activate the transitive closure of a composite object. Such methods can be used at the discretion of the client. However, the dereference operator is always available as a means of activating a persistent object in a uniform manner.

3. Object activation

The power of the persistent derefence operation lies in the capability for transparently extending the behavior and properties of the persistent OOP. The primary behavior is loading an object from secondary storage and enabling it as a computational agent. The primary property of the referent is uniqueness.

Principle: Persistent OOPs encapsulates all information necessary to enable the referent.

Principle: Persistent OOPs reference objects with unique identity.

Again, principles are expressed from the perspective of the OODB client. The encapsulation principle asserts that, as far as the client is concerned, the dereference operation is sufficient of itself to enable the persistent object to respond to messages. Likewise, the uniqueness principle asserts that persistent OOPs act *as if* they reference objects with unique identity. They may in practice be local identifiers relative to some unique context. An offset in a virtual memory page may be one such local identifier. The set of all databases for a given OODBMS product may be one such context.

The uniqueness principle says nothing about the format of the persistent identifier. The format of persistent identifiers and their mapping to virtual memory is a key product differentiator. Uniformity of persistent identifier format is best decided in the marketplace rather than a standards committee.*

We call the procedures associated with an object its *type code.* This is broadly construed to include database procedures attached as attributes to an object (triggers). The seamless integration of an OODB with the host language applies to all type code.

Principle: All type code is expressible in the reference language.

---

* Object transport may be an exception, and is not considered here.

This does not preclude the adoption of specialized language features in certain type code bindings. Rather, it states that wherever type code can be bound to an object, it can take the form of a reference language construct.

The distributed, heterogenous processing environment of most OODB applications requires particular care in setting the boundaries of the DBMS execution model. The notion of an application as a self-contained executable image is replaced by cooperative interaction among independent computational agents. These agents may be objects managed by different DBMSs. Traditional DBMSs make quite liberal assumptions about their domain of control. OODBMSs, on the other hand, should respect object boundaries. An executing DBMS should gracefully surrender control upon dereference of a persistent object managed by a different DBMS. This suggests a standardized approach to dynamic type code binding.

Principle: The database execution model only governs the lexical scope of embedding methods.

The principle essentially asserts that OODB execution models should not conflict in ways that jeapordize persistent object encapsulation. This principle is particularly important for an integrated application framework such as a design environment or hypertext navigator. Divergent solutions to hetereogeneous database access-- to wit the vendor-specific, ad hoc database "switches" of the relational DBMS community-- are not appropriate in the decentralized, distributed processing environment of object-oriented databases.

4. Database objects

Where the programming language is wanting in features that support persistent data management, the DBMS architect must supply them. The options available to developers for specifying these features include:

- Providing a library of functions: e.g., `store()`, `startTransaction()`.

- Declaring classes with the desired feature: e.g., a `PersistentObject` class.

- Extending the reference language: e.g., a `persistent` storage class qualifier for variable definition.

Functionality is added to objects by adding methods to classes. Global functions are an aberration that violate the spirit of object orientation. Database functionality is no less bound to this principle.

Principle: Encapsulate type–specific functionality.

Encapsulations are appropriate wherever a well–defined database object can be identified. Transactions come to mind in this regard. The reification of transactions as coordinator objects demonstrates the usefulness of this principle. Drawing transactions into the realm of first–class objects leverages the capability of the host language. A few examples:

- o A transaction object has scope (where it can be referenced) and extent (when it can be referenced) according to the rules of the language, rather than the restrictive and arbitrary rules of the DBMS.

- o Transaction objects encapsulate appropriate behavior (begin, commit, etc.) that can be extended by clients.

- o Parameterized types (e.g., set and graph) support long and nested transactions in a natural way.

- o Transaction constructors can be extended to explicitly identify parent transactions.

5. Language extension

OODB extensions to the reference language are, by definition, non–standard. This carries hazards to the OODB client, not the least of which is reliance on a single source of database services. Language extensions are, however, a simple and elegant approach to adding features that are orthogonal to type. Arguably, persistence is one such feature.

Principle: Limit language extensions to type–orthogonal features.

We can strengthen this principle as follows: Implement type–orthogonal features with language extensions. However, this is unadvisable; we illustrate this with the persistence feature. There are two reasonable means of declaring persistence: 1) a language extension, and 2) a property of a persistent class. While the merits of the two approaches are debatable, OODB standardization is not (yet) an appropriate forum for this debate. This is an area of creative research where product differentiation yields competitive advantage. It is counterproductive at this point to standardize an approach to declaration of persistence.

6. Summary

We have considered database features of persistent object access and suggested several principles of OODB development. The principles seek to further the goals of encapsulation, integration with object–oriented languages, and distributed database support. We have suggested areas of potential standardization. The task of arriving at a consensus on the form of these standards is left as open problem for this forum.

Notes toward a Standard Object-Oriented DDL and DML

THOMAS ATWOOD
JACK ORENSTEIN

Object Design, Inc., Burlington, Massachusetts (617) 270-9797
tom@odi.com
jack@odi.com

INTRODUCTION

The effort to define standards in database management systems and programming languages has historically come after there was some considerable body of commercial experience. If standards in object-oriented software follow the same pattern, then we are still very early in the cycle for the formal proposal of standards. However, there is at least one factor that suggests that the transition to OO data management may occur faster than the transition from CODASYL to relational database management systems. The transition from CODASYL to relational database management systems was a relatively 'narrow' phenomenon, one which affected only the DBMS world. The transition from relational to object-oriented data management is part of a broad-based transition to object-oriented software that is occuring in user interfaces, programming languages (Smalltalk, the Common Lisp Object System (CLOS), Object Pascal, C++, ...), object-oriented design methodologies and object oriented data management. The software development community's conviction that the benefits of OO programming are real is generating pressure to standardize OO programming languages, and then in turn OO database management systems, so that ISVs and in-house development organizations at large companies feel that they can safely take advantage of this new technology. Pressure for standards is therefore emerging earlier in the life cycle of the OO technology than it did in the life cycle of relational technology. Database management standards are getting pushed by the fact that software developers have made/are making the transition to OO programming languages, and want a compatible data management substrate.

Although it may therefore be premature to propose specific data models or DML syntax, we think it is time to articulate some framing perspectives. There are, to our way of thinking, broadly two ways in which OODBMS standards can be approached: the *minimalist position*, and the *OO-DBPL position*. The minimalist position argues that although we may define a fairly rich conceptual model, we map into each programming language only as much of that model as the programming language's type system and control structures can handle. Consider C++ to give the discussion some concreteness.

Assume that the OO conceptual model that the OODBTG eventually agrees on is a superset of the entity-relationship-attribute (ERA) model which has become common as a basis for conceptual design tools. We replace E with O for object, and add Op for operation, yielding a model which has as its primitives Objects, Relationships between objects, Attributes of Objects, and Operations. The C++ object model is not this strong. Its fundamental assumption is that object types (classes) are defined by behavior alone - i.e. operations. An object in C++ is a data structure encapsulated by a set of operations. There is no (abstract) notion of attributes of objects, or relationships between them. The best the C++ class definer can do is to define a pair of operations get_x and set_x for each missing attribute or relationship. (Making x public, and manipulating x directly is possible, but violates encapsulation.) The minimalist position would map only the operation portion of the OO-DBMS conceptual model into C++. The OO-DBPL position would restrict the use of 'data members' to the 'private' (implementation) part of the class definition and would introduce attributes and relationships as formally distinct members of the public part of the class definition.

Our crystal ball at this early stage in the discussions says that although the OO-DBPL solution is conceptually the cleanest, the pragmatic politics of change may leave us somewhere between the two positions. In this note we would like to clarify the minimalist position a bit.

CHOICES MADE BY THE RELATIONAL GENERATION

If we step back far enough to gain some perspective, the relational generation seems to have made two mistakes in its definition of query languages:

1. It attempted to define a single language (SQL) to meet the needs of the interactive non-professional user and the professional programmer. It proposed the same language, SQL, as an interactive query language and as a data manipulation sublanguage for programming languages

2. It forced a single DML (again SQL) into different programming languages without regard for the syntactic structure the host language.

The problem with the first is that the needs of the programmer and the interactive end user are very different. A language which attempts to satisfy the requirements of both, ends up satisfying the requirements of neither. And that is in fact what has happened. SQL is rarely actually used directly as an interactive language that the end-user types in. Instead a series of graphical query-building interfaces and menu-based application specific interfaces have been developed. The end user

interacts with these and they generate SQL. SQL has become an intermediate representation language that is generated by a program — something analagous to Pcode generated by a compiler — rather than something the end user interacts with directly.

The problem with the second, forcing a single syntactically constant DML into several programming languages, is that different programming languages have very different syntax. Attempting to force a single, uniform DML into all of them means that it will fit well with none of them. And worse, the consequence of the first decision — to use SQL as both a query language and a DML — was that SQL had to be a fairly complete language. It had its own notions of variables, expressions, and statements (e.g, 'GROUP BY ... HAVING'). And these, by definition, clashed with those of the programming languages in which SQL is embedded. Figure 1 illustrates the situation. The shaded areas represents the areas of clash.

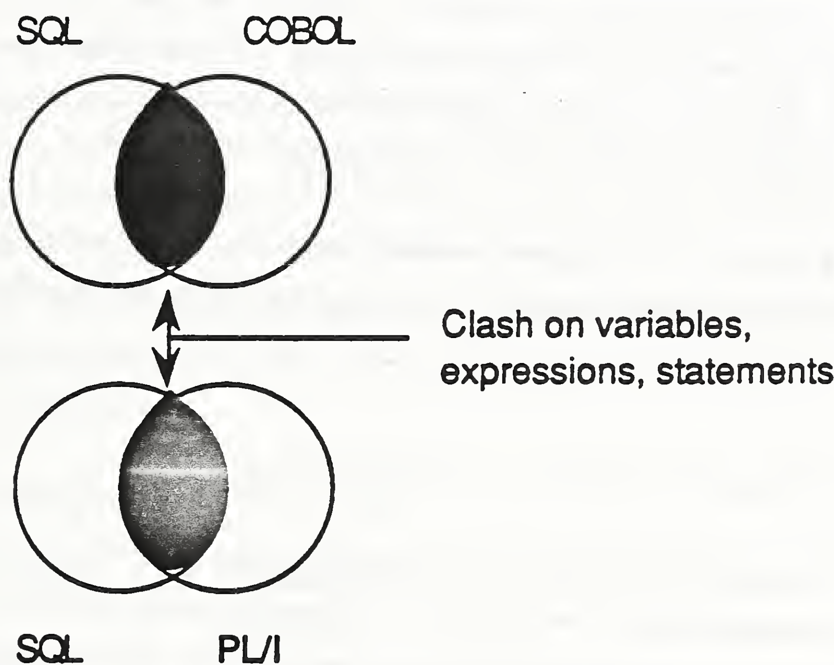


Figure 1

This may have made sense for programmers who had to write PL/I in the morning and COBOL in the afternoon; at least the SQL stayed the same. And it may also have had some rationale in the era of big mainframes with batch compilation and 24 hour turn around on compile/link/test cycles; the programmer could test his database access routines interactively, before committing them to the compiled program. But most programmers write in a single language, and the era of batch compilation has passed.

CHOICES FOR THE OBJECT GENERATION

We suggest that any OO-DBMS standards effort make the opposite choices on both of these issues.

1. The syntax of the interactive query language (OQL/I for want of a better name) should *not* be the same as the syntax for the DML embedded in programming languages, and

2. Instead of forcing a single relatively complete DML sublanguage into each programming language irrespective of its syntactic clash with the programming language, we should attempt to maintain a common Conceptual Model and map its DML into each programming language using a syntax appropriate to that language and using the existing control statements, expressions, defined by the programming language to the greatest extent possible.

The result will be a series of DML sublanguages which require only a small set of new constructs for any given programming language and 'feel right' to the programmer using that language.

THE MINIMALIST APPROACH FOR C++ — AN ILLUSTRATION

The series of code fragments discussed in the remainder of this paper illustrate some of the specifics of how this approach might be realized for the programming language C++. The examples demonstrate possible syntax for creating persistent objects, getting/setting attribute values, traversing relationships, asking queries, and breaking programs into transactions.

As a basis for illustrating these capabilities, we introduce the C++ declaration for the class PROFESSOR.

```
class PROFESSOR : public PERSON
{
public:
    // attributes
    int office_extension;
```

```
        // relationships
        DEPARTMENT* department;
        set <COURSE*> courses_taught;
        set <STUDENT*> advisees;
    };
```

This declarations of courses_taught and advisees use the parameterized type capability proposed for C++.

OBJECT CREATION

C++ allows the programmer to create objects with either of two lifetimes — a lifetime limited to the life of the procedure in which the object was initially created, or a lifetime which is coterminous with the process in which the object was created. It has no support for creating objects which outlive the process. Support for persistent objects which outlive the process in which they were created can be easily added to each of the two constructs for object creation supported by C++: (i) automatic creation of objects on entry into the scope of the declaration by the program's flow of control and (ii) explicit programmer control over object creation. The syntactic forms used by C++ and the extended forms are illustrated below:

automatic allocation on entry to scope

syntax	lifetime	allocated in	allocated by
`PERSON p;`	procedure	stack	PL runtime*
`static PERSON p;`	process	heap	PL runtime
`persistent PERSON p;`	database	database	DBMS runtime

* PL: programming language

dynamic allocation

syntax	lifetime	allocated in	allocated by
`PERSON* p = new PERSON;`	process	heap	PL runtime
`PERSON* p = new(db)PERSON;`	database	database	DBMS runtime

Here, db is used as an argument to new to indicate that the object being allocated should be placed in the indicated database. This relies on the C++ "placement" argument to new. In this approach, lifetime is orthogonal to type - it is a property of individual objects, not of object types, and any type may have both ordinary transient instances, and persistent instances.

ATTRIBUTES

Getting and setting the value of attributes requires no syntactic additions; it is done using the dot and arrow notation used for data members of a class together with standard C++ assignment statements:

```
PROFESSOR* p;
...
int x = p->office_extension;
p->office_extension = 345;
```

RELATIONSHIPS

One to one relationships are declared using standard C++ data member syntax. 1:N and M:N relationships are declared using a form of the proposed C++ parameterized type notation, and a set of built-in aggregate classes. For example, the 1:N advisor/advisee relationship between a professor and his advisees is declared by two mirror declarations, one on the class PROFESSOR, the other on the class STUDENT:

```
class PROFESSOR : public PERSON
{
   ...
   set <STUDENT*> advisees inverse advisor;
   ...
};

class STUDENT :public PERSON
{
```

```
    ...
    PROFESSOR advisor inverse advisees;
    ...
};
```

The DBMS runtime automatically maintains the inverse relationship. That is, if for a given student s you assign a professor p to his advisor field,

```
s->advisor = p;
```

then the system automatically adds the student s to the set of advisees of p.

Creating and deleting individual relationships between one professor and one student is done using set insertion/removal syntax.

```
p->advisees += s;  // insert s into the set of
                   // students who are p's advisees.

p->advisees -= s;  // remove s from the set of
                   // students who are p's advisees.
```

In each case, the set update causes maintenance of the relationship by updating the advisor attribute of s.

Even the associative retrieval syntax can be a fairly natural extension of C++'s syntax for retrieving particular elements of an array — array [element#] , e.g., a[i]. In C++ the identifier a is assumed to refer to an array. In the proposed C++ DML we allow it to refer to an instance of any collection class (formally any subclass of the class collection): set, bag, list, ... The array subscript allows selection only by ordinal position. We expand it to allow associative retrieval expressions which have the full power of OQL queries, i.e., they can select qualifying objects based on their attribute values and/or on the relationships they participate in with other objects.

```
// Find Prof. Guttag.
p = PROFESSORS[ name == "Guttag" ];

// Find the teacher of epistemology
p = PROFESSORS[ courses_taught[ title == "epistemology ]];
```

The first query locates the professor whose name is Guttag and returns it as the result of the query. The second query locates the professor whose courses_taught set includes the course whose title is epistemology

Finally, to handle what has come to be called the 'impedence mismatch' between set at a time query languages and one-at-a-time programming languages, the AT&T Bell Labs OO-DBMS group [Agrawal 89], introduced a "foreach" iterator that seems to provide a simple solution. Example:

```
foreach (STUDENT* s, p->advisees)
   <statement>
```

This construct executes the statement once for each binding of the loop variable s to a student that is an advisee of p. The statements within the loop can be any C++ statements. There is no limitation that they be only query language statements. Similiarly, the programmer can freely intermix query language variables and programming language variables in the query expression and does not have to worry that only one or the other is acceptable in some contexts.

TRANSACTIONS

A concept which has historically been absent from programming languages is the notion of transactions (although there is some overlap with the notion of exception handling). A nested transaction model with lexical scoping of transactions might be introduced into C++ as follows:

```
transaction(<args>)
{
   transaction (<args>)
   {
      ...
      if (minor_disaster) transaction::abort;

      if major_disaster transaction::top_level_abort;
   }
}
```

All of the statements within a transaction block are again standard C++ statements. If exception handling becomes part of the language, exceptions raised within the scope of a transaction but caught by handlers outside would have to be carefully implemented to undo any nested transactions until they got out to the level of the handler, but that should all be taken care of by the language implementor,

not the application programmer. Dynamic transaction begin could be introduced without any syntactic extension at all. TRANSACTION could be defined as a class which supported the operations Begin, Commit, Abort, ... Again, the point is that database functionality can be introduced in a way which fits smoothly into the base programming language, and in fact, takes advantage of the control statements and scope delimiters available within that language to keep the data manipulation sublanguage to a minimum.

CONCLUSION

The sum total of the DML sublanguage constructs which in this minimalist proposal have been embedded in C++ are:

- The key word `persistent`,
- an `inverse` clause on relationship declarations,
- associative retrieval expressions, and
- the `transaction` statement.

The result is a database programming environment very familiar feeling to the C++ programmer. Diagramatically, we get a series of programming language specific DMLs that look like figure 2 rather than figure 1:

Figure 2

This notion of a language-specific syntactic binding is one which is independent of where the task group lands along the Minimalist <-> OO-DBPL spectrum. We raise it as an issue which deserves consideration before we delve into the syntax wars.

To the extent that strong interest in OO data management is emerging first in market segments moving from C to C++, we expect interest in DML standards to mature first within the context of C++. A variant of the C++ DML illustrated above has been implemented as part of the C++ DML preprocessor for our own firm's OO-DBMS product ObjectStore. The same tack could be taken with Ada or CLOS.

Object Design is making a commitment to be an active contributor to the discusssion and evolution of a draft standard through the ANSI process. The authors may be reached for comment at the above address.

A Model for OODB Queries*

Dave D. Straube
M. Tamer Özsu

Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1
{daves,ozsu}@cs.ualberta.ca

Abstract

Variations on what constitutes an object-oriented database are as numerous as the problems to which these databases are being applied. Given such diversity, standardization efforts should define a framework which allows many specific models to be defined. We outline a flexible framework for defining the query component of an object-oriented database and provide a sample model of queries which fits within this framework.

1 Introduction

It has been said that the nice thing about standards is that there are so many of them to choose from. This overabundance of standards may be due to the fact that the problems we wish to solve are not standard at all. In fact, they are so diverse that we need a standard for each of them. Object-oriented databases (OODB) have emerged in response to problems which are not well served by traditional database models and as a result, exhibit the same diversity as the problems they address. One could conclude, then, that developing an OODB standard will reduce the applicability of object-oriented databases and actually inhibit their acceptance and use. To avoid this dilemma, we propose that any OODB standard really be a meta-standard; a template within which any of the many viable OODB database models can be constructed.

We claim that enough contradictory views of what constitutes an object-oriented database exist to require the meta-standard approach. For example, the following are just a few OODB features which have caused considerable controversy and over which there is no consensus:

*This research has been supported in part by the National Sciences and Engineering Research Council (NSERC) of Canada under operating grant OGP-0951.

- Objects can be viewed as instances of abstract data types whose representation and implementation are hidden or as complex data structures whose components are available for inspection and modification.
- Each object is an instance of only one class or can be a member of multiple classes.
- Classes can inherit from only one parent (single inheritance) or can inherit from any number of parents (multiple inheritance).
- The database schema defines one hierarchy for both implementation and behavioral inheritance or may allow implementation to be inherited from classes other than those which specify inherited behavior.
- Equality operators ignore object structure (e.g. identity equality) or have the ability to compare structures to some depth.

Each of these tradeoffs presents a divergent view of what constitutes an OODB and strong arguments may be made in favor of any of them. It may be that the only way to resolve the conflicting views is to base the specific of the data model on the application domain involved. Thus, for example, the object model for a CAD/CAM database may be different than the object model for an office information system database. Furthermore, it should be stated that the tradeoffs that we have specified above are based on our understanding of the issues **today**. It is difficult to claim that the solutions developed to date (which form the basis of the existing tradeoffs) are the only ones that will ever be generated or that they are the ones that will stand the test of time. In the absence of a consensus over these tradeoffs, or even on the set of alternative solutions, the reasonable path to follow at this time would be to define a meta-standard which specifies what features should be included in an object-oriented database management system.

Our research program on object-oriented database systems has concentrated on a formal investigation of query processing issues. In this paper we identify query processing related issues which a standard should address, followed by a brief overview of the query model we are using in our research. The query model should be viewed as an example how some of the design tradeoffs can be addressed and what the interrelationships between the issues are.

2 Query Model Design Issues

We have defined a query processing methodology for an OODB (Figure 1) similar to that for relational systems. (see, for example[7, 9]). Queries are expressed in a declarative language which requires no user knowledge of object implementations, access paths or processing strategies. The query expression is first reduced to a normalized form and then converted to an equivalent object algebra expression. This form of the query is a nested expression which can be viewed as a tree whose nodes are algebra operators and whose leaves represent the extents of classes in the database. The algebra expression is then checked for type consistency to insure that predicates and methods are not applied to objects which do not support the requested functions [17]. This is not as simple as type checking in general programming languages since intermediate results, which are sets of objects, may be composed of heterogeneous types. The next step in query processing is the application of equivalence preserving rewrite rules [8, 16] to the type consistent algebra expression. Lastly, an access plan which takes into account object implementations is generated from the optimized algebra expression.

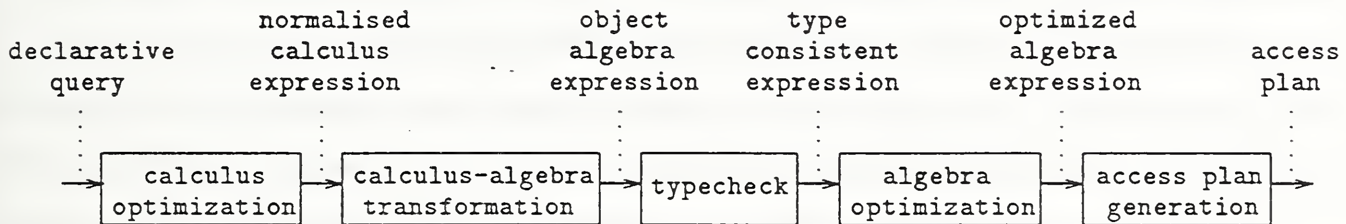


Figure 1: Query processing methodology

Definition of the query processing methodology in this way enables us to define the following design tradeoffs:

What is an object? The query model must fully describe the visible components of objects which can be accessed by query operations. For example, if objects are tuple-valued as in [3], then query expressions can directly access tuple fields by name.

What are the primitive object operators and their semantics? Examples of primitive object operators are the various equality operators such as $=_O$ (identity equality) and $=_i$ (i-equality) of [14] and record field selectors of [3]. These operators are used to construct predicates and complex query expressions. Another common operator in object-oriented databases is that for applying a method to an object [15].

What data values have a textual representation in the query language? Regardless of the level of abstraction maintained by objects, the query language must have some way of expressing literal values. The syntax may permit representation of only atomic values corresponding to types predefined in the database or may include complex value constructors such as those of [12] for constructing aggregates and set values.

What is the query language? Query languages can range from formal calculi and algebras to ad-hoc extensions to a query language or an object-oriented programming language. Any translation between equivalent query forms should be identified.

How are predicates formed? Predicates are typically formed by combining primitive object operators with boolean connectives and existential and universal quantifiers. However, user defined quantifiers such as *majority* [8] and aggregate operators such as *sum* or *count* [6] may also be allowed.

What is the input to a query? The two most common query inputs are the objects represented by some type(s) in the database or a user defined collection of objects. Another possibility is that the query input is a (sub)schema of the database as in [1].

What is the result of a query? Query results are usually sets of objects, or, as in the case of [1], a subschema which denotes a set of objects. An important issue here is whether queries are *object preserving* or *object creating* [13]. Object preserving queries always return a subset of the query input while object creating queries return new objects which are not part of the original database. This impacts whether queries can be used as views or compared using the identity operators.

How are queries type checked? Type checking queries is desirable since it identifies errors early and without the potentially harmful results which could occur at run time. Furthermore, almost all applications have the requirement that a program variable be iteratively bound to consecutive elements of a query result. The query and application language type checking mechanisms must be compatible for this binding to be performed in a type safe manner [17].

3 Query Model

This section presents the query model we are using to investigate query processing issues in object-oriented databases. The model indicates how some of the tradeoffs can be resolved and can also

serve as a meta-standard since it can easily be extended. For example, the algebra operators that are defined (Section 3.5) are object preserving. The algebra, however, can easily be extended to include object creating operators. The description in this paper is relatively intuitive. For a rigorous and formal definition of these concepts, the reader is referred to [16].

3.1 Objects

Objects are viewed as instances of abstract data types (ADT) which can only be manipulated via functions defined by the type. Types are organized in an inheritance hierarchy which allows multiple inheritance. Each object has a unique, time invariant identity which is independent of its state. Relations on object identities such as equality and set inclusion provide the basis for query primitives which qualify algebra operators. All other relations among objects are implemented by the ADT interfaces.

3.2 Classes and Methods

In accordance with popular object-oriented terminology, a *class* defines both an ADT interface via *methods* and stands for all the objects which are instances of the type. Methods are named functions whose arguments and result are objects. Each method has a signature of the form $C_1 \times \ldots \times C_n \rightarrow C_{result}$ where $C_1 \ldots C_n$ specify the class of the argument objects and C_{result} specifies the class of the result object. All classes in the database form a lattice where the root node represents the most general class of objects and any individual class may have multiple parents. Subclasses inherit behavior from their parents and may define additional methods. Thus, the class lattice provides inclusion polymorphism [4] which allows an object of class C to be used in any context specifying a superclass of C [14].

3.3 Primitive Object Operations

Objects encapsulate a state and a behavior. Methods defined on the class which an object is an instance of define the object's behavior. Behavior is revealed by applying a method to an object. The result of a method application is another object. The dot notation $<o_1 \ldots o_n>.m_1.m_2 \cdots m_m$ is used to denote method application and method composition. Assuming methods m_1 and m_m take three arguments each, and method m_2 takes 2 arguments, then Figure 2 illustrates the processing denoted by this operation. Method m_1 is applied to objects $<o_1, o_2, o_3>$ resulting in object r_1, method m_2 is applied to objects $<r_1, o_4>$ returning object r_2, and so on until the final result object

r_m is obtained by applying method m_m to objects $<r_{m-1}, o_{n-1}, o_n>$. $<o_1 \ldots o_n>$.*mlist* will be used when the list of method names is unimportant.

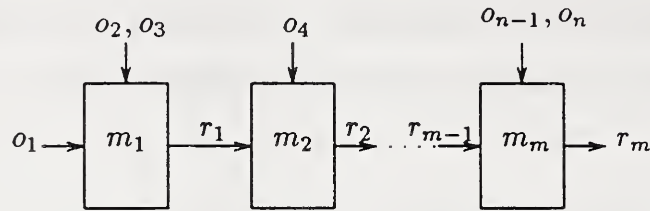


Figure 2: Composition of method applications.

An object's state is captured by its value which is distinct from its identity [10, 15]. Object values are either an *atomic value* provided by the database system (int, string, uninterpreted byte sequence [5]), a *set value* which is a collection of object identifiers, or a *structural value*. Structural values are visible only to class implementors and can encompass attributes (tuples), discriminated unions, etc. as in [2]. Any aspects of structural values which are required by users of a class should be revealed by the implementor via a method.

We define four comparison operators which can be used in queries: $==$, $\in$, $=_{\{\}}$ and $=$ whose semantics are shown in Tables 1 and 2. The $==$ operator tests for object identity equality; i.e. $o_i == o_j$ evaluates to true when o_i and o_j denote the same object. The $\in$ and $=_{\{\}}$ operators apply to set valued objects and denote set value inclusion and set value equality respectively. As shown in the tables, one of the operands can denote a value if required. The last operator, $=$, can only be used to test the value of an atomic object.

Table 1: Semantics of $o_i \theta o_j$ as a function of the object value type.

		$o_i \theta o_j$		
o_i	o_j	$==$	$\in$	$=_{\{\}}$
atomic	atomic	T/F	undefined	undefined
	structural	T/F	undefined	undefined
	set	T/F	T/F	undefined
structural	atomic	T/F	undefined	undefined
	structural	T/F	undefined	undefined
	set	T/F	T/F	undefined
set	atomic	T/F	undefined	undefined
	structural	T/F	undefined	undefined
	set	T/F	T/F	T/F

Table 2: Semantics of $a\theta o_i$ as a function of the object value type.

		$a\theta o_i$		
a	o_i	$=$	$\in$	$=_{\{\}}$
val_1	atomic	T/F	undefined	undefined
	structural	undefined	undefined	undefined
	set	undefined	T/F	undefined
$\{val_1, \ldots, val_n\}$	atomic	undefined	undefined	undefined
	structural	undefined	undefined	undefined
	set	T/F	T/F	T/F

3.4 Predicate Formation

Atoms are primitive operations of the data model which return a boolean result. Atoms reference lower case, single letter object variables which range over sets of objects when used in a query. The legal atoms are as follows:

- $o_i \theta o_j$ where:
  - o_i and o_j are object variables or denote an operation of the form $<o_1 \ldots o_n>.mlist$ where $o_1 \ldots o_n$ are object variables.
  - θ is one of the operators $==$, $\in$ or $=_{\{\}}$.
- $a\theta o_i$ where:
  - o_i is an object variable or denotes an operation of the form $<o_1 \ldots o_n>.mlist$ where $o_1 \ldots o_n$ are object variables.
  - a is the textual representation of an atomic value or a set of atomic values.
  - θ is one of the operators $=$, $\in$ or $=_{\{\}}$.

Predicates are formed by connecting atoms with $\wedge$, $\vee$ and $\neg$ as required.

Example 3.1 Let p, q and r be object variables. Then the following are examples of legal atoms and their semantics:

1. $(p == q)$ – Are the objects denoted by p and q the same object?
2. $(p \in <q, r>.mlist)$ – Is the identifier of p contained in the set value of the object obtained by applying the methods in *mlist* to the objects $<q, r>$?

3. ($<p,q>.mlist =_{\{\}} r$) – Is the set value of the object obtained by applying the methods in *mlist* to the objects $<p,q>$ pairwise equal to the set value of the object denoted by r?

4. (“59” $= p$) – Is the atomic value of the object denoted by p “59”?

5. (“59” $\in p$) – Does the set value of the object denoted by p include an identifier for the object whose atomic value is “59”?

6. (“59, 61” $=_{\{\}} <p,q,r>.mlist$) – Does the set value of the object obtained by applying the methods in *mlist* to the objects $<p,q,r>$ contain only two identifiers for objects whose atomic values are “59” and “61”? ◇

3.5 Query Language – An Object Algebra

The object algebra contains both binary and n-ary operators. Let Θ be an operator in the algebra. We will use the notation $P \ \Theta \ \langle Q_1 \ldots Q_k \rangle$ for algebra expressions where P and Q_i denote sets of objects. In the case of a binary operator we will use $P \ \Theta \ Q$ without loss of generality. The algebra defines five object preserving operators: union, difference, select, generate and map. These are fundamental operators; other may be defined (e.g. intersection) for convenience in terms of these.

Union (denoted $P \cup Q$): The union is the set of objects which are in P or Q or both. An equivalent expression for union is $\{ \ o \mid P(o) \vee Q(o) \ \}$.

Difference (denoted $P - Q$): The difference is the set of objects which are in P and not in Q. An equivalent expression for difference is $\{ \ o \mid P(o) \wedge \neg Q(o) \ \}$. The intersection operator, $P \cap Q$, can be derived by $P - (P - Q)$.

Select (denoted $P \ \sigma_F \ \langle Q_1 \ldots Q_k \rangle$): Select returns the objects denoted by p in each vector $<p, q_1 \ldots q_k> \ \in \ P \times Q_1 \times \ldots \times Q_k$ which satisfies the predicate F. An equivalent expression for select is $\{ \ p \mid P(p) \wedge Q_1(q_1) \wedge \ldots \wedge Q_k(q_k) \wedge F(p, q_1 \ldots q_k) \ \}$.

The select is similar to, but more powerful than, that of [14] which allows only one operand. Multiple operands permit explicit joins as described in [11]. An explicit join is a join between arbitrary classes which support (a sequence of) method applications resulting in comparable objects.

Generate (denoted $Q_1 \ \gamma_F^t \ \langle Q_2 \ldots Q_k \rangle$): F is a predicate with the condition that it must contain one or more generating atoms for the target variable t, i.e. t does not range over any of

the argument sets. The operation returns the objects denoted by t in F for each vector $<q_1 \ldots q_k> \in Q_1 \times \ldots \times Q_k$ which satisfies the predicate F. An equivalent expression for generate is $\{ \, t \mid Q_1(q_1) \wedge \ldots \wedge Q_k(Q_k) \wedge F(t, q_1 \ldots q_k)) \, \}$.

Generating atoms are unique in that they generate values for variables which do not range over an input set of the query (Table 3). They are called generating atoms because they generate objects for x from a constant value (entry 5), from the content of other objects (entries 2,4), or by applying methods to objects (entries 3,4). As an illustration, consider the query $Q \; \gamma^{p}_{(p \in <q,r>.mlist)} \; \langle R \rangle$. Variables q and r range over the argument sets Q and R respectively and thus can be considered 'bound' in the query. However, variable p is not bound to any argument set and the atom $p \in <q,r>.mlist$ will evaluate to true only when p ranges over the objects in the set value of the objects obtained by the method applications. Under these conditions then, the atom generates values for p.

Table 3: Generating atoms for x.

1	$x == o$
2	$x \in o$
3	$x ==<o_1, \ldots, o_n>.mlist$
4	$x \in<o_1, \ldots, o_n>.mlist$
5	$x = a$

Map (denoted $Q_1 \;\; \mapsto_{mlist} \;\; \langle Q_2 \ldots Q_k \rangle$): Let *mlist* be a list of method names of the form $m_1 \cdots m_m$. Map applies the sequence of methods in *mlist* to each object $q_1 \in Q_1$ using objects in $\langle Q_2 \ldots Q_k \rangle$ as parameters to the methods in *mlist*. This returns the set of objects resulting from each sequence application. If no method in *mlist* requires any parameters, then $\langle Q_2 \ldots Q_k \rangle$ is the empty sequence $\langle \; \rangle$. Map is a special case of the generate operator whose equivalent is $\{ \, t \mid Q_1(q_1) \wedge \ldots \wedge Q_k(q_k) \wedge t ==<q_1 \ldots q_k>.mlist \, \}$. This form of the generate operation warrants its own definition as it occurs frequently and supports several useful optimizations. Map is similar to the **image** operator of [14], except that it is not restricted to unary methods.

References

[1] A. Alashqur, S. Su, and H. Lam. OQL: A Query Language for Manipulating Object-Oriented Databases. In *Proc. 15th International Conference on Very Large Databases*, pages 433–442,

1989.

[2] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[3] J. Banerjee, W. Kim, and K. Kim. Queries in Object-Oriented Databases. In *Proc. 4th Int'l. Conf. on Data Engineering*, pages 31–38, February 1988.

[4] L. Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[5] M. Carey, D. DeWitt, J. Richardson, and E. Shetika. Object and File Management in the EXODUS Extensible Database System. In *Proc. 12th International Conference on Very Large Databases*, pages 91–100, August 1986.

[6] C. J. Date. *A Guide to the SQL Standard.* Addison Wesley, June 1987.

[7] G. Gardarin and P. Valduriez. *Relational Databases and Knowledge Bases.* Addison Wesley, 1989.

[8] W. Hasan and H. Pirahesh. Query Rewrite Optimization in Starburst. Technical Report TR RJ 6367, IBM Alamden Research Center, August 1988.

[9] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):112–152, June 1984.

[10] S. N. Khoshafian and G. P. Copeland. Object Identity. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 406–416, September 1986.

[11] W. Kim. A Model of Queries for Object-Oriented Databases. In *Proc. 15th International Conference on Very Large Databases*, pages 423–432, 1989.

[12] S. L. Osborn. Identity, Equality and Query Optimization. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 346–351. Springer Verlag, 1988.

[13] M. Scholl and H. Schek. A Relational Object Model. Unpublished manuscript.

[14] G. Shaw and S. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proc. 6th Int'l. Conf. on Data Engineering*, pages 154–162, February 1990.

[15] M. Stefik and D. Bobrow. Object-Oriented Programming: Themes and Variations. *The AI Magazine*, pages 40–62, 1985.

[16] D. Straube and M. T. Özsu. Queries and Query Processing in Object-Oriented Database Systems. *Submitted to ACM Transactions on Information Systems.* Also available as University of Alberta Computing Science technical report TR 90-11, April, 1990.

[17] D. Straube and M. T. Özsu. Type Consistency of Queries in an Object-Oriented Database System. *Submitted to the Object-Oriented Programming Systems and Languages Conference*, 1990.

Intelligent SQL

by

Setrag Khoshafian

1. Introduction

The database management system technology of the 1980s, namely relational databases, fell far short of providing the necessary abstraction or model to act as the repository of the hypermedia objects and applications of the 1990s. They lacked several fundamental components to support integrated applications:

(1) Applications in the 1990s will increasingly use multi–media data types: long text fields, raster images, vector images, voice data, animation, video input, to name just a few. Relational databases provided very limited support for these data types. In the more advanced relational databases of the 1980s the multi–media data types as stored as long un–interpreted strings of bits. Most of them do not have direct support for access methods, associative retrieval or updates on these fields.

(2) The relational databases of the 1980s could not provide direct and natural representation of graph structured object spaces. The object spaces in relational systems are flat tables. Hence complex relationships between multi–media objects and products are not directly expressible.

(3) The querying languages of relational databases (most notably the SQL standard) are not expressive enough. Relational algebra is not computationally complete. For instance it is impossible to evaluate the transitive closure of a relation using SQL. Relational database also lack declarative inferencing rules.

The deficiencies of relational databases are being solved by the next generation database management systems, namely intelligent databases. A comprehensive definition and exposition of intelligent databases is given in the book *Intelligent Databases* (Parsaye et al., 1989), by Wiley publications. As defined there intelligent databases are:

> *Databases that manage information in a natural way, making that information easy to store, access, and use.*

The emphasis in intelligent databases is on information rather than data, because intelligent database incorporate not only traditional applications such as inventory management, but also knowledge bases, automatic discovery systems, imaging applications, and so on.

Another important concept is the naturalness of the intelligent database model. This natural representation of the knowledge and information in intelligent databases expresses itself in:

(1) Strong support of the "natural" multi–media data types: text, image, voice, animation, video, etc.

2) An object oriented database model which allows a more direct representation of real world models,

(3) The support of declarative rules to express directly and naturally semantic relationships amongst objects.

Intelligent databases represent a new technology for information management that has evolved as a result of the integration of traditional approaches to databases with more recent fields such as:

· Object–oriented concepts

· Expert systems

· Hypermedia

· Information retrieval

· Distributed Databases

This is illustrtated in Figure 1. The object–oriented capabilities in Intelligent SQL include *abstract data typing*, *inheritance,* and *object identity*. Each of these features are discussed in more details in subsequent sections.

In addition, expert systems are integrated in Intelligent SQL through a tight and seamless integration of rules and SQL. Intelligent SQL incorporates full–text searches on text attributes (columns). The user–interface of Intelligent SQL is a object–oriented hypermedia environment. For more details on intelligent hypermedia interfaces to databases see (Parsaye et al., 1989). Another very important component of Intelligent SQL is the distributed database capabilities of Intelligent SQL. Through Intelligent SQL users can *introduce* foreign and remote databases to the Intelligent SQL engine and use these foreign tables like local data. The distributed database constructs of Intelligent SQL

will not be discussed in this paper.

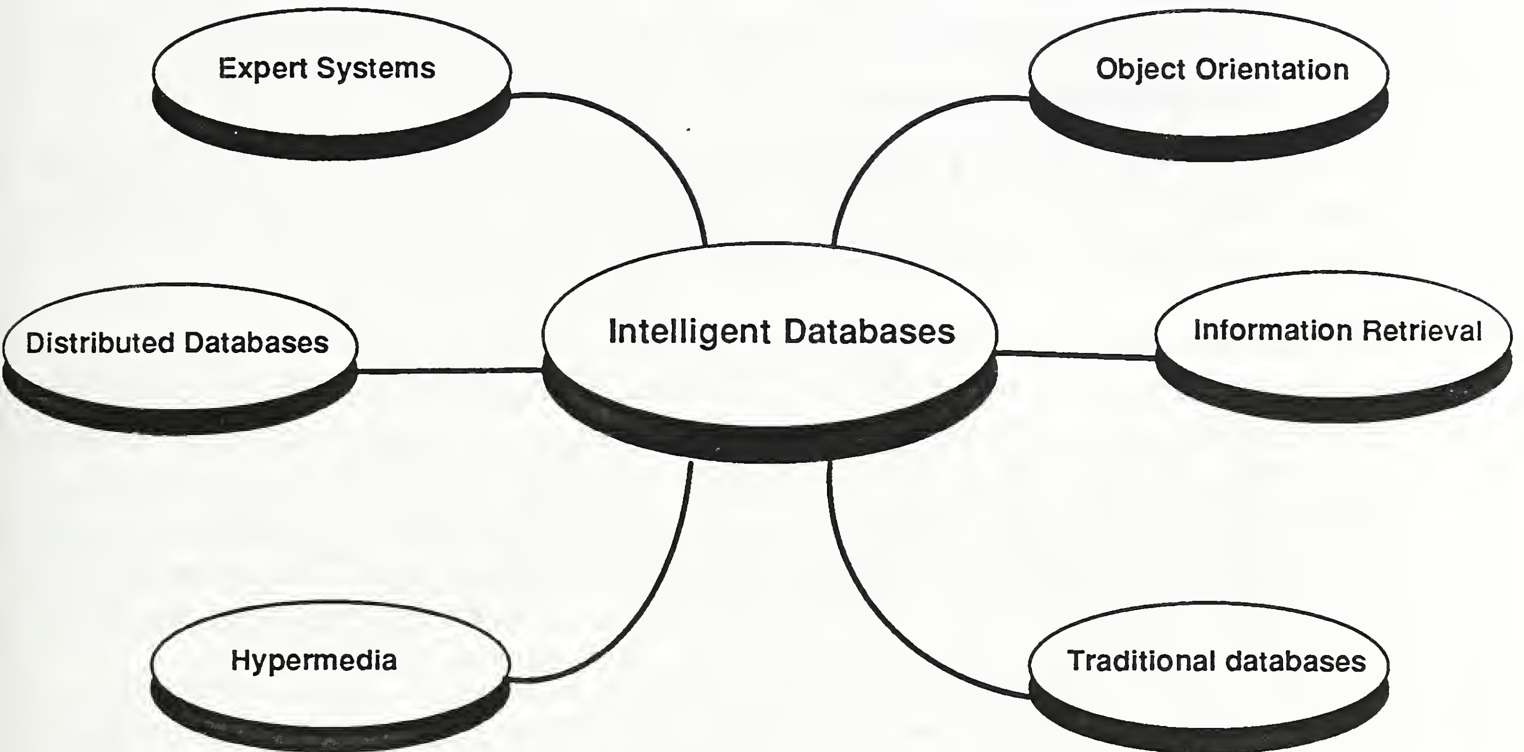


Figure 1: The Technologies Integrated in Intelligent Databases

1.2 The Three Layered Architecture

Intelligent databases thus represent the evolution of a number of distinct paths of technological development. Until recently, these technologies were treated in isolation, with each technology being only weakly linked to others. For instance, expert systems have relied on little more than file–transfer protocols to gather data from databases. Due to the phenomenal growth in each field, the connections and correspondences to the other fields did not have time to form. Now that these technologies have reached a stage of maturity, it is possible to define an overall unifying structure for viewing all these fields as

parts of a blueprint for intelligent databases. Intelligent databases provide a common approach to the access and use of information for analysis and decision making.

The top–level architecture of the intelligent database consists of three levels:

- High level tools
- High level user interface
- Intelligent database engine

As Figure 2 shows, this is a staircase layered architecture. Users and developers may independently access different layers at different times. The details and functionality of each level is given in (Parsaye et al., 1989). Here we present a brief overview.

The first of these levels is the high level tools level. These tools provide the user with a number of facilities: intelligent search capabilities, data quality and integrity control, and automated discovery. These high–level tools represent an external library of powerful tools. Most of these tools may be broadly classified as information management techniques, similar to spreadsheets and graphic representation tools. They look and work much as their stand–alone equivalents, but they are modified so as to be compatible with the intelligent database model. They are object oriented, and their basic structure mirrors the object representation methods of the intelligent database model.

The second level is the high–level user interface. This is the level that users directly interact with. This level creates the model of the task and the database environment that users interact with. It has to deal as much with how the user wants to think about databases and information management as with how the database engine actually operates. Associated with this level are a set of representation tools that enhance the functionality of the intelligent database.

The user interface is presented in two aspects. First, there is a core model that is presented to the user. This core model consists of the object oriented representation of information, along with a set of integrated tools for creating new object types, browsing among objects, searching, and asking questions. In addition, there is a set of high–level tools, which enhance the functionality of the intelligent database system for certain classes of user.

The base level of the system is the intelligent database engine and its data model. This model allows for a deductive object oriented representation of information, which can be expressed and operated on in a variety of ways. The engine includes backward– and forward–chaining inference procedures, as well as optimizing compilers, drivers for the external media devices, and version handlers.

In the rest of the paper, we concentrate on the intelligent database engine and its interface: Intelligent SQL. As mentioned earlier, the intelligent database engine incorporates a model

that allows for a deductive object–oriented representation of information that c
expressed and operated on in a variety of ways. The engine includes backward
forward– chaining inference procedures as well as drivers for the external media de
version handlers, optimizing compilers, and the like.

The intelligent database engine with its deductive object oriented data modeling sup
the underlying repository for the integrated applications and products which use th
level tools and high level user interfaces. The engine is the core and the most imp
component of future systems. The intelligent database engine provides the functi
and the performance for supporting the integrated applications. In addition, access tc
databases, distributed inferencing and database management systems are ach
through the intelligent database engine.

Intelligent database engines consist of two components:

(1) The Deductive Object Oriented Data Model Interface: This provide
interface to the layers above. DOOD is basically the core intelligent dat
language. Applications which intend to use the intelligent database e
submit programs written in the DOOD language, which is a dialect of SQL,
Intelligent SQL.

(2) The actual engine which compiles, optimizes and executes DOOD prog
Figure 2 indicates some of the components of the architecture: The Opti
The Inference Engine, The Meta–Information Manager, The Trans
Manager, The Multi–Media Data Manager, and The Storage Manager.

The rest of the paper is dedicated to a more detailed of the *object–oriented* capabili
Intelligent SQL

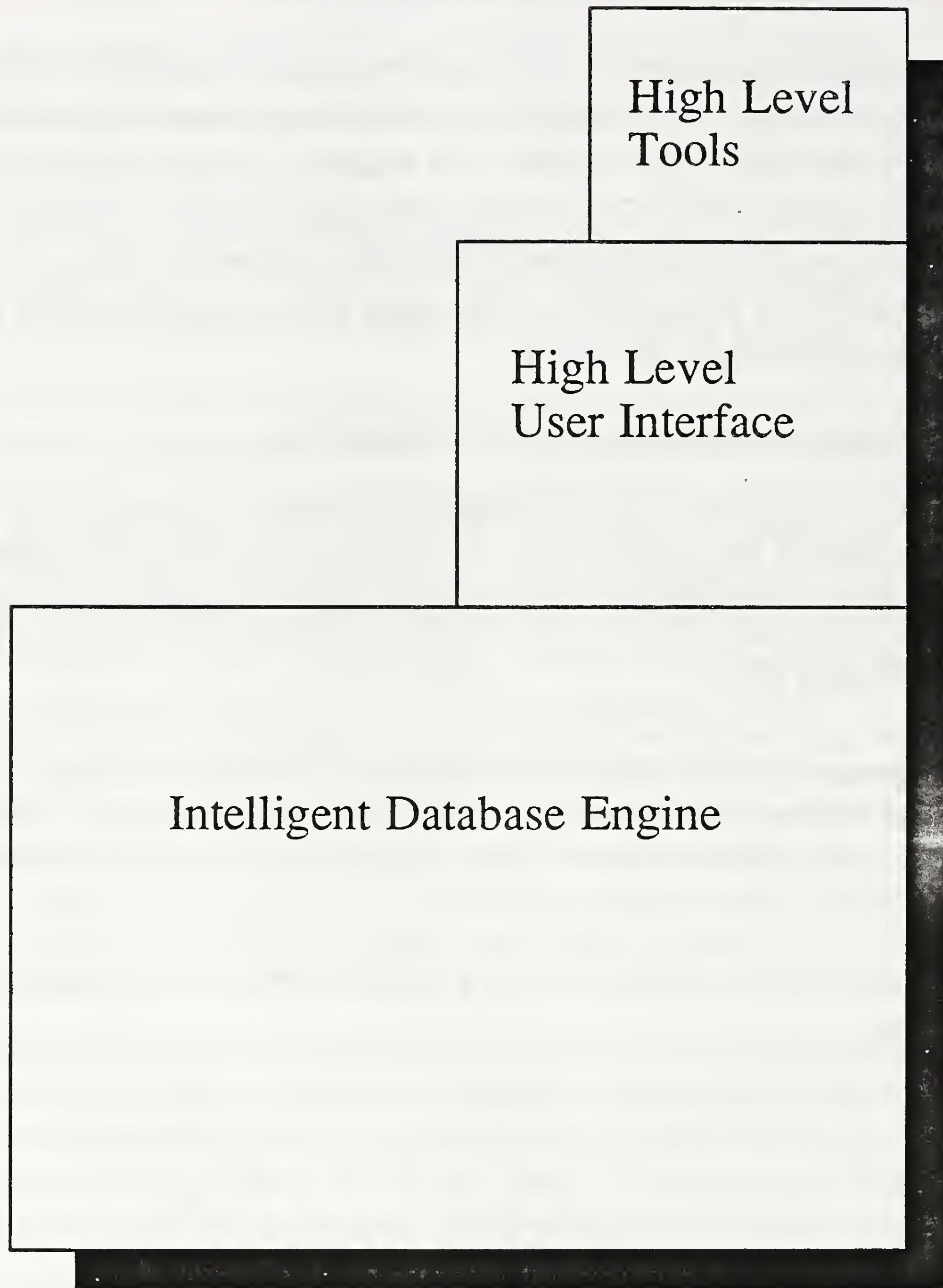


Figure 2: The Three Layered Architecture

1.2 Why Yet Another Dialect of SQL?

The intelligent database strategy is to capture and represent the deductive, object oriented, multi–media capabilities of intelligent databases as a dialect of the most popular database language, namely SQL. The dialect is called Intelligent SQL. This will provide a natural migration path from existing relational technology.

Extending SQL with full compatibility with the SQL standard provides the most popular and useful introduction of intelligent database concepts in database management systems for the following reasons:

(1) SQL is the most popular relational query language and is endorsed by almost all the major developers of relational systems. Therefore, an extended and upward compatible SQL can easily encompass the application programs developed in SQL, and thus maintain the investment in these programs.

(2) It is the only relational language which has a standard developed for it.

(3) SQL is also being promoted as the interface language of database engines, such as the Ashton–Tate/Microsoft SQL Server and the Oracle OS/2 Server. Therefore, new applications developed in the extended SQL can easily call these database servers for both efficiency and access of remote data.

(4) SQL improves upon "navigational" models such as the hierarchical and network models, in that it is more declarative. SQL provides higher level database definition and manipulation constructs allowing the programmers to specify what they want from a DBMS (vs telling the system how to obtain and manipulate the database).

Supporting powerful language extensions to model the real world as closely as possible is just the interface of an intelligent database engine. To be usable, the additional functionalities must be supported natively in the underlying engine, to guarantee higher performance. Furthermore, the incorporation of the SQL extensions should not penalize the less sophisticated users of "vanilla" SQL. The intelligent database engine described here is driven by these performance and efficiency requirements. In this paper we primarily concentrate on the object–oriented capabilities of Intelligent SQL. A description of the AI (Artificial Intelligence) and IR (Information Retrieval) capabilities of Intelligent SQL is given in (Khoshafian et al., 1990) and (Khoshafian and Abnous, 1990).

3. Encapsulation

One of the most important features of object orientation is encapsulation. Encapsulation is an abstraction mechanism whereby an object is accessed and modified only through

external interface routines and functions or operations (typically known as "methods" in the object oriented jargon). The internal implementation details, data structures, and storage elements used to implement the object and its operations are not visible to the clients accessing and manipulating the object. The object's behavior is fully defined through the set of abstract operations defined for the objects.

The object "types" in SQL are:

(a) Built–in atomic object types, usually known as "column types" in SQL jargon. An SQL implementation will provide a number of built–in atomic types such as Character, integer, Decimal, Logical, or Date.

(b) The rows or tuples which constitute the elements of the tables. The values of the attributes (columns) in a row can only be atomic.

(c) The tables (relations) of homogenous tuples (rows).

3.1 User Defined ADTs

Perhaps the most obvious and easiest extension of SQL to incorporate object orientation would be the support of user–defined abstract data types. In other words allow the "base types" of SQL (integer, decimal, char, etc.) to be extensible with additional types, whose behavior is encapsulated in methods (operations) written in some high level language such as C.

Similar attempts include (Osborn and Heaven 1986), POSTGRES (Stonebraker, 1986), and FAD (Bancilhon et al. 1987). In FAD, the routines defining the abstract data types are written in C. This same technique is incorporated in Intelligent SQL.

```
CREATE ADT-CLASS <adt-name> [<adt parameters>] (
        INSTANCE VARIABLES ( <instance-variable declaration>
                        [,<instance-variable declaration>);
        OPERATIONS (
                <operation-description> ( <parameter declaration>
                                        [, <parameter declaration>]
                        <operator-body>
         [, <operation-description> ( <parameter declaration>
         [,<parameter declaration>]
                        <operator-body>] ) )
```

where <operation–description> indicates the operator's name, the language, and signature (we assume all operations are functions which return values):

```
<operation-description> ::= <operation name> <operation signature>
```

The <operation signature> specifies the types of the input and output parameter of the operation. Each operation is actually a "function" (although operations could have "side effects" updating the internal state of the object).

Both parameter and instance variable declarations contain the name and type of the variables. The same operator could be overloaded as described in Section 2.2. The convention is that the first argument determines the operator's implementation (and hence its binding). Therefore, the first parameter of the parameter declaration list should always be the same as the ADT being defined. The instance variable pertain to the first parameter and can be accessed and updated within an operator body.

The instance variables are accessed by prefixing the instance variable's name by the object's handle, delimited through a ".". Thus if O is an object of ADT type T with instance variable X, then X is obtained through O.X. The instance variable types are either built–in or user defined ADTs.

For example, we can define the class Stack as follows:

```
CREATE CLASS Stack (
        INSTANCE VARIABLES ( ARRAY StArr[50] of INTEGER,
                        Top     INTEGER,
                        Size    INTEGER);
        OPERATIONS ( Push
                        Stack X INTEGER -> Stack
                        ( St Stack, Value Integer
                        St.StArr[St.Top] = Value
                        St.Top = St.Top + 1
                        RETURN St);
                Pop
                        Stack -> INTEGER
                        (St Stack
                        St.Top = St.Top - 1
                        RETURN StArr[St.Top + 1]);
...))
```

The operations defined in different user–defined abstract data types could be *overloaded.* In conventional languages such as C or Pascal the same operation could apply to different object types depending on the context. For example the "+" operator in 11 + 5 is integer addition; in 1.1 + 5.0 it is floating point addition; in 'aaa' + 'ccc' it is string concatenation.

The overloading of operations (methods) is a powerful concept. To illustrate its power assume we have a heterogeneous stack with 1 to Top elements and an overloaded "Print" operation for each type. Then with an object oriented approach the code to print all the elements of the stack would be:

```
for i:= 1 to Top do
        Print(St[i])
```

With a more conventional approach we would have to encode the type, have as many "Print" operations as there are types (e.g. PrintString, PrintInteger, etc.) and a huge "case" statement to invoke the appropriate print for the object type.

3.2 Parametric Polymorphism

Overloading is an "ad–hoc" polymorphism (Cardelli and Wgner 1985), in the sense that the overloaded operators are totally unrelated. Another powerful object oriented concept is parametric polymorphism. In the Stack example we specified the maximum number of elements for the stack (50) and the types of the elements of the stack (INTEGER).

Using this approach we would need a separate declaration for each combination of stack size and stack element type. Parametric polymorphism solves this problem by parametrizing types and boundaries. These parameters must be bound at compile time.

For example the parametrized declaration of stacks would be:

```
CREATE CLASS Stack[Max, EType] (
        INSTANCE VARIABLES ( ARRAY StArr[Max] of EType,
                        Top     INTEGER,
                        Size    INTEGER);
        OPERATIONS Push
                        Stack X EType -> Stack
                        ( St Stack, Value Integer
                        St.StArr[St.Top] = Value
                        St.Top = St.Top + 1
                        RETURN St);
        Pop
                        Stack -> EType
                        (St Stack
                                St.Top = St.Top - 1
                                RETURN StArr[St.Top + 1]);
```

As we said earlier, when we declare an object to be a stack we must bind the Max and Etype values. For example we can create a table of Accounts, where Payables is a stack of dollars:

```
CREATE TABLE Accounts(
        AcountNumber            INTEGER,
        Location                CHAR(20),
        Payables                Stack[10, DOLLAR])
```

3. Tuple Valued Attributes

To motivate the importance and appropriateness of tuple valued attributes consider a simple example: the address of persons. Addresses have structure: House/Apparment Number, Street, City, State, Zip, Country, etc.

Thus it is very natural to have a "tuple" (record, struct) valued attribute which stores the address of a person or a company. With the current "relational" approach we have the following choices for storing the address:

(1) Store the address as a character string and let the "user" store and retrieve individual fields of the address. With user–defined abstract data types the user can define methods to retrieve each individual address field. However, this alternative introduces an almost arbitrary partitioning between the user defined types and the types directly supported by the database management systems. Rows of tables are tuples. There is no reason why these tuples cannot be values of attributes (columns).

(2) Have AddressStreetNumber, AddressCity, AddressState, etc. fields in the Persons table. This is not natural since the semantics of the association between the address fields are lost. Furthermore if the Persons relation has several such "tuple" valued attributes, its schema will become incomprehensible.

(3) To alleviate some of the problems in (2) store the addresses in a separate table with a primary key, and for each Persons row store the address foreign key. Although this strategy is "cleaner" than (2), accessing the address fields will involve a join.

The problem with all these strategies stems from the underlying problem of not being able to have "type completeness".

Thus we introduce "tuple" valued attributes and an extended "dot" notation to retrieve the field values of nested tuples.

To declare a tuple, we can either create and name a tuple type, or directly declare an attribute to be of type TUPLE (<tuple attribute declarations>).

To create a names tuple type the syntax is:

```
<tuple definition> ::= CREATE TUPLE <tuple name>
                              ( <tuple elements> )
<tuple element> ::= <column definition>
<column definition> ::= <column name> <data type>
               | <column name> <tuple declaration>
<tuple declaration> ::= TUPLE ( <tuple elements> )
```

For example we can create a tuple valued Address attribute either through:

```
CREATE TUPLE ADDRESS (
               Number         INTEGER,
               StrtName       CHAR(20),
               AptNumber      INTEGER,
               City           CHAR(20)
               State          Char(20),
               Zip            INTEGER)

CREATE TABLE Persons (
               NAme           CHAR(20),
               Age            INTEGER,
               HomeAddr       ADDRESS)
```

Alternatively we can have:

```
CREATE TABLE Persons (
               NAme           CHAR(20),
               Age            INTEGER,
               HomeAddr       TUPLE(
                               Number         INTEGER,
                               StrtName       CHAR(20),
                               ApTNumber      INTEGER,
                               City           CHAR(20)
                               State          Char(20),
                               Zip            INTEGER)
               )
```

3.1 Extended dot "." Notation

In SQL columns are referenced by their name either directly or, if there is an ambiguity, prepended by a table name. The value of a column (attribute) in a row is a base type. Since

in the proposed extension the value can be a row (tuple), we must have a way to columns of this row directly.

The extended dot notation would modify the 'column–ref' production to:

```
column-ref ::= [column qualifier .] column-sequence
column-sequence ::= column
                  | column-sequence . column
```

Thus to retrieve the city in which Jerry lives we have:

```
SELECT HomeAddr.City
FROM Persons
WHERE Name = "jerry"
```

Note that a nested tuple can contain other nested tuples. For example the City a itself be a tuple indicating the name, population, elevation, meyer, et.c of the ci can have

```
CREATE TUPLE CITY (
               Name                    CHAR(20),
               Population              INTEGER,
               Meyer                   CHAR(20),
               Elevation               FEET)
```

We declare the attribute type of City in Address to be CITY:

```
CREATE TUPLE ADDRESS (
               Number          INTEGER,
               StrtName        CHAR(20),
               AptNumber       INTEGER,
               City            CITY,
               State           Char(20),
               Zip             INTEGER)

CREATE TABLE Persons (
               Name            CHAR(20),
               Age             INTEGER,
               HomeAddr        ADDRESS)
```

Jerry's address's street name and city's meyer's name could be retrieved thr

```
SELECT HomeAddr.StrName, HomeAddr.City.Meyer
FROM Persons
WHERE Name = "jerry"
```

As we shall see in the next section, the extended dot notation is also used to indicate table options on nested tuples when creating tables.

3.2 Tuple Expressions

The extended dot notation is used to retrieve tuple or scalar valued attributes of table rows. *Tuple expressions* is n additional construct for assigning and seraching with tuple valued attributes. The general form of a tuple expression is:

```
TUPLE(<value list>)
```

where the <value list> contains a list of scalar and/or tuple expressions seperated by commas. For example:

```
TUPLE(1322, "Spring", NULL,
            TUPLE("San Simon", 10000, "Billy Joe",
                  20000))
```

is a tuple expression whihc can appear in a data manipulation statement (as demonstrated in the next section) or in a search condtion as in:

```
SELECT Name, Age
FROM Persons
WHERE HomeAddr = TUPLE(1322, "Spring", NULL,
                 TUPLE("San Simon", 10000, "Billy Joe",
                       20000))
```

Tuple expressions can also be used in other DML (data manipulation language expressions) For example we can modify the entire address of John Smith through:

```
UPDATE Persons
SET Address = TUPLE(1322, "Spring", NULL,
                TUPLE("San Simon", 10000, "Billy Joe",
                      20000))
WHERE Name = "john Smith"
```

Similarly the Intelligent SQL INSERT statement also allows tuple expressions to be inserted into tbales which have tuple valued attributes. For example we can have:

```
INSERT
INTO Persons (Name, Age, HomeAddr)
VALUES ("John Smith", 22,
                        TUPLE(1322, "Spring", NULL,
                         TUPLE("San Simon", 10000, "Billy Joe",
                                   20000))
```

4. Object Identity

A powerful object oriented concept is the incorporation of the strong notion of object identity (Khoshafian and Copeland 1986) in a database language. Identity is that property of an object which distinguishes it from all other objects. Most programming and database languages use variable names to distinguish objects, mixing addressability and identity. Most database systems use identifier keys (i.e. attributes which uniquely identity a tuple) to distinguish objects, mixing data value and identity. Object–oriented languages employ separate mechanisms for these concepts, so that each object maintains a separate and consistent notion of identity regardless of how it is accessed, what it contains (i.e. its value), or how it is modeled with descriptive data (Khoshafian and Copeland 1986).

The importance and power of object identity has been recognized and incorporated in numerous database languages, including functional languages based on semantic data models (Shipman 1981), LDM (Kuper and Vardi 1985), and more recently FAD (Bancilhon et al. 1987).

If a language supports the strong notion of identity, it is not necessary for the user to "see" or "manipulate" the object identity directly. In addition to special operators which manipulate object identity, in interactive environments users are given "handles" to an object. Using these handles users can traverse graphically structured object spaces. In the sequel we shall incorporate these special operators and handles in SQL.

There are three types of objects in SQL:

(a) Built–in atomic domain types, usually known as "column types" in SQL jargon. These include Character, Small Integer, Integer, Decimal, Numeric, Float, and Data.

(b) The rows or tuples which constitute the elements of the tables. The values of the attributes (columns) in a row can only be atomic.

(c) The tables (relations) of homogenous rows.

In Phase I we implement the extension of SQL which incorporates identity to rows of base table, tuple valued attributes, and atomic (user defined ADT) attributes only. Phase II will

generalize object identity in SQL to incoporate "own" references and identities for set valued attributes.

In incorporating object identity we have a clean underlying object model which is based on the identity based model of FAD.

4.2 Indetity in Intelligent SQL

The main extension in the data (or schema) definition language of SQL is the provision for an object id of a row from a table as a column data type, or a tuple type, or a user–defined abstract data type. Thus in the extended SQL BNF we'll have:

```
<column definition> ::=
                <column name> <data type>
              | <column name> <tuple declaration>
              | <column name> OBJECT <object type>

<object type> ::=
                <data type>
                <tuple declaration>
                ROW <table-name>
```

Note that if we do not have the keyword OBJECT we will get actually a *value* of the indicated type (e.g. a tuple). Thus if we have a Department table declared as:

```
        CREATE TABLE Departments(
                DeptName        CHAR(20),
                DeptNum         INTEGER,
                Budget          FLOAT(2))
```

Then, assuming each employee works in exactly one department, we can declare an Employee table as

```
        CREATE TABLE Employees(
                EmployeeName            CHAR(20),
                EmployeeAge             INTEGER,
                EmployeeSalary          INTEGER,
                EmployeeRank            INTEGER,
                EmployeeDept            OBJECT ROW
                                        Departments)
```

With object identity the attribute or column value of a row is a tuple. Hence the xtended dot notaion can also be used here to access the attributes of the nested tuples. So with the

schema of EMployees and Departements we can retrieve the department name of Jo'es department throughL

```
SELECT EmployeeDept.DeptName
FROM Employees
WHERE EmployeeName = "Joe"
```

As far as retrieval is concerned, extended dot natation is basically what we need.

As we mentioned earlier, there are three types of equality associated with object identity. To simplify the support of identity, in the initial version of Intelligent SQL we shall overload the predicate "=" to mean:

(1) *identical* if the two arguments are OBJECTs

(2) *Shallow Equal* if the two arguments are tuples

(3) Comparison of atomic values, otherwise

For instance to rerieve the names of all the employees who work in the same departmet as Joe we have:

```
SELECT AllEmps.EmployeeName
FROM Employees AllEmpls, Employees JoeEmps
WHERE JoeEmps.EmployeeName = "Joe" and
   AllEmps.EmployeeDEPT = JoeEmps.EmployeeDEPT
```

For updates, we need a mechanism to have the same object be referenced by multiple parent objects. To this end we introduce an ASSIGN operation whose general form is:.

```
ASSIGN <object exprssion> TO <object expression>
```

where is <object expression> a select statemetn identifying a single object:

```
<object expression> ::= <selext statement>
```

This means the <select clause> of the <select statement> is a concatenation of attribute names seperated by "."s. For instance to ASSIGN the address of Mary to Joe we have:

```
ASSIGN SELECT Address From Persons WHERE Name = "Joe"
TO
SELECT Address From Persons WHERE Name = "Mary"
```

There must be *one* Joe and *one* Mary in Persons

5. Inheritance For ADTs, Tuples, and Tables

The third powerful object oriented concept to be integrated in SQL is inheritance.

In object oriented systems and languages, using inheritance we can build new classes or types on top of an existing less specialized hierarchy of classes (types), instead of re–writing the new classes which resemble existing classes in many aspects, from scratch. The new classes inherit from existing classes both representation and behavior (i.e. operators or methods).

Similar to encapsulation and object identity, inheritance is also a natural model of the real world. We are used to abstracting and classifying information in inheritance hierarchies. Thus we think of mammals as a sub–class of vertebrates, and dogs as a sub–class of mammals, and Yorkshire terriers as a sub–class of dogs.

inheritance allows us to re–use and share the code of the less specialized classes. It provides us with perhaps the most natural and efficient way of organizing complex system and application code. As we shall show this provides with the cleanest mechanism for organizing large application programs.

Inheritance in SQL

Next we illustrate how we introduce inheritance to dBASE/SQL. In dBASE/SQL we have the "base" (atomic) types which are extensible through user defined abstract data types as discussed in Section 2. We also have two type constructors: tuples and tables. In Section 3 we showed how tuple types could be created (very similar to the creation of tables).

As far as inheritance is concerned we shall incorporate the following:

(1) Inheritance of user defined ADT's allowing specialization and overloading of representation and/or operations

(2) Inheritance for tuple types

(3) Table inheritance through specialization hierarchies.

Next we detail the strategies, syntax, and semantics for each of these types.

5.1 Inheritance in ADTs

The syntax to indicate the super–classes of an ADT subclass is through the construct

```
ADT-SUPER-CLASS <super-class list>
```

Thus the production of an ADT declarations becomes:

```
CREATE ADT-CLASS <class name> (

 [ADT-SUPER-CLASS <super-class list>]
 [INSTANCE VARIABLES AS <inherited instance var list>]
 [OPERATIONS AS <inherited operations list>] ]
    INSTANCE     VARIABLES     (<instance-variable     declaration>
[,<instance-variable declaration>]);
        OPERATIONS(<operation-description>(
                            <par-declaration list>
                            <operator-body>
[,<operation-description>(
                            <par-declaration list>
                            <operator-body> )] )
```

In terms of the semantics of ADT inheritance we follow the following conventions:

(1) The inherited instance variables are "visible" to the inheriting (sub–class) clients of the ADT (but not the "instantiating" clients of the ADT).

(2) An ADT class can "specialize" a super–class through either:

   (a) Declaring additional instance variables

   (b) Declaring additional operations

   (c) Specializing (overriding) existing instance variables as described in (4).

   (d) Specializing (overriding) existing operations as described in (4).

(3) If an AS clause is not specified then if an ADT class inherits from more than one super–class and there is a conflict either in the representation (instance variables) or behavior (operations) than the left–most classes in the ADT–SUPER–CLASS declaration take precedence.

(4) If an AS clause is specified for either the instance variables or the operations, then *all* the instance variables of the super–classes are inherited and given the names specified in the AS clause, in left to right order.

(5) If a sub–class overrides an instance variable of a super–class than the type of the instance variable in the sub–class must be a subtype of the type of the overriden instance variable.

5.2 Inheritance in Tuple Classes

Similar to inheritance in user defined ADT's we can have (multiple) inheritance of tuple types through the construct:

```
TUPLE-SUPER-TYPE <super-type list>
```

Thus the tuple definition becomes:

```
<tuple definition> ::=
                    CREATE TUPLE <tuple name>
                    [TUPLE-SUPER-TYPE <super-type list>
                     [ATTRIBUTES AS <inherited attribute list>]]
                                   ( <tuple elements> )
```

In terms of the semantics of tuple inheritance we follow the following conventions:

(1) A tuple type can "specialize" a super–type through either:

(a) Declaring additional attributes

(c) Specializing (overriding) existing attributes as described in (4)

(2) If an AS clause is not specified then if the tuple type inherits from more than one super–type and there is a conflict (i.e. an attribute with the same name is declared in more than one super–type) than the left–most tuple types in the TUPLE–SUPER–TYPE declaration take precedence.

(3) If an AS clause is specified then *all* the attributes are inherited from the super–types and re–named with the names given in the <inherited attribute list>, in left to right order of the super types.

(4) If a tuple type overrides an attribute of a super–type than the type of the attribute in the sub–type must be a subtype of the type of the overriden attribute.

As a simple example for tuple inheritance, consider business address as a sub–type of addresses:

```
CREATE TUPLE ADDRESS (
                    Number          INTEGER,
                    StrtName        CHAR(20),
                    AptNumber       INTEGER,
                    City            CHAR(20)
                    State           Char(20),
                    Zip             INTEGER)

CREATE TUPLE BUSINESS-ADDRESS
          TUPLE-SUPER-TYPE ADDRESS
          (OfficeNo                 INTEGER)
```

Similarly we can have a foreign address:

```
CREATE TUPLE FOREIGN-ADDRESS
        TUPLE-SUPER-TYPE ADDRESS
        (Country                CHAR(20))
```

And a foreign business address as:

```
CREATE TUPLE FOREIGN-ADDRESS
 TUPLE-SUPER-TYPE BUSINESS-ADDRESS, FOREIGN-ADDRESS
```

5.3 Inheritance for Tables

There are two mechanisms for creating a sub–table of an existing tables;

(1) *Specialization through Additional Column Extensions*: where the sub–table introduces additional columns, specializing the super–table.

(2) *Specialization through Restrictions*; where the sub–table either restricts the domains of attributes or introduces arbitrary restrictions on the attributes of a table

In fact we can *combine* extension and restriction specializations and declare a table T2 to be a sub–table of a table T1, where T2 introduces *additional* attributes to T1 *and* restrict the domain of the existing attributes of T1.

Next we shall define the syntax and semantics of both extensions and restrictions.

5.3.1 Additional Column ("Horizontal") Extensions

For table inheritance with additional column extensions we follow an SQL 3 proposal. More specifically the CREATE TABLE construct has the following production:

```
CREATE TABLE <table name>
                <table elements>
                | <sub-table clause> [<table elements>]
```

where:

```
<subtable clause> ::=
        SPECIALIZES <super-table name list>
        [AS <column name list>]
```

where:

```
<table attribute list> ::= <table attributes> , <table attributes> ...

<table attributes > ::= <table name> (<column name list>)
```

In terms of the semantics of table inheritance, we have the following rules:

(1) Each table name in the super–table list is the name of a table

(2) The graph of the table inheritance hierarchies is actually a DAG (directed acyclic graph): in other words there are no cycles.

(3) If tables in the super–table list have a common predecessor (i.e. they transitively inherit from a common root table), then there will be *one* copy of the shared columns. If shared columns have different names then there must be an AS clause which specifies the name used in the table beung created.

(4) If the AS clause is not specified, the table inherits from more than one super–table and there is a conflict (i.e. unrelated column with the same name is declared in more than one super–table), the left–most table in the sub–table clause takes precedence. Note that if the conflicting columns come from the same predecessor rule (3) will still be valid since there will be one copy of the shared columns.

(5) If we have an AS clause, then the column names from the inherited tables are re–named.

(6) Whenever we insert/delete/modify a tuple in a table which inherits from other tables, the sub–tuples corresponding to the super–tables are (logically) inserted/deleted/modified in the super–tables. More formally, the inheritance hierarchy has a set inclusion semantics. This means if T1 is a sub–table of T2, then T2's extension (i.e. the set of all rows in T2) *will also contain the elements of T1*. A table as a set contains all its sub–tables as sub–sets.

(7) Whenever we delete/modify a tuple in a super–table, the tuples corresponding to the deleted/modified tuple in all sub–tables are deleted/modified.

Examples

As a simple example consider the tables Students, Staff, and ResearchAssistants:

```
CREATE TABLE Persons (
                Name            CHAR(20),
                Age             INTEGER,
                HomeAddr        ADDRESS)

CREATE TABLE Staff
        SPECIALIZES Persons
        (       ID              CHAR(10),
                OfficeAddress   BUSINESS-ADDRESS)
PRIMARY KEY(ID)

CREATE TABLE Students
```

```
        SPECIALIZES Persons
        (       ID              INTEGER,
                Advisor         OBJECT ROW Staff)
PRIMARY KEY(ID)

CREATE TABLE ResearchAssistants (
        SPECIALIZES Students, Staff AS Name, Age, HomeAddr,
                        StaffID, OAddr, StudentID, Adv
        (       ResearchArea    CHAR(20)
```

With this definition the ResearchAssistants table will be initially empty. As discussed in Section 5.3.3, unique constraints on StaffID and StudentID are inherited. Whenever we insert a row in ResearchAssistants, we can SELECT or modify that row from Staff or Students.

5.3.1.1 SELECTs on Super–Tables

The table inheritance has a *set inclusion semantics.* More specifically, the table Persons includes not only the elements of Persons but also the elements of all the sub–tables of Persons (of course *without* the additional attributes – i.e. the projections of these tables on the inherited attributes). Thus when we execute:

```
SELECT *
FROM Persons
```

we retrieve the elements of Persons (who are *not* Staff or Student), as well as the elements of Staff, Student, and ResearchAssistants projected on (Name, Age, HomeAddr).

5.3.2 Restriction Specializations ("Vertical" Modifications)

In some cases tables are specialization of super–tables because of certain *restrictions* which are relevant only to the more specialized sub–table. Therefore through introducing restrictions on one or more columns of a table we can create a subset sub–set specialization of the table. For example in the table Persons we can create the sub–table Adults which have *all* the attributes of Persons, except that the age attribute is in the range greater or equal to 21.

There are two ways for indicating restrictions:

(1) Through specifying a WHERE clause

(2) Through indicating the sub–domain of inherited columns

Syntactically for restriction inheritance we introduce a WHERE clause to indicate the appropriate restrictions on the columns of a super–table. Thus the table creation clause becomes:

```
CREATE TABLE <table name>
                <table elements>
                | <sub-table clause> [<table elements>]
```

where:

```
<subtable clause> ::=
        SPECIALIZES <super-table restricted name list>
        [AS <column name list>]

<super-table restricted name list> ::= <restricted table name> , ...

<restricted table name> ::=
                table name> [<subtable restriction clause>]

<subtable restriction clause> ::=
        WHERE <restriction clause>

<table attribute list> ::= <table attributes> , <table attributes> ...

<table attributes > ::= <table name> (<column name list>)
```

In terms of semantics we have the following additional rules;

(14) If a sub–table restriction clause clause is specified for a super–table then it indicates the intension that all tuples satisfying the predicate are automatically elements of the sub–table. Actually subtables which are defined through restrictions *could have non empty intersections.*

(15) if a sub–table restriction clause clause is specified and the table also incorporates column extensions then whenever a tuple in a super–table satisfies the constraint then that tuple is also an element of the sub–table with the additional columns values set to DEFAULT or NULL.

(16) If a sub–table restriction clause clause is specified and there are more than one tables in the super–table name list and there are columns of some tables which are not joined with columns of other tables then whenever a tuple in a super–table satisfies the constraint then that tuple is also an element of the sub–table with the additional columns values set to DEFAULT or NULL.

(17) In lieau of (15) and (16) with a sub–table restriction, each column which could potentially be set to null due to super–table tuples which satisfy the constraint must

either have a default clause or, otherwise, must not have a NOT NULL constraint.

(18) If OUTER is specified then the sub–table restriction is unnecessary and will be ignored.

For example we can create an Employees table and a sub–table which contains the highly paid employees:

```
CREATE TABLE Employees (
                EmpName          CHAR(20),
                EmpAge           INTEGER,
                Salary           INTEGER
                Department       INTEGER)
PRIMARY KEY(EmpName)

CREATE TABLE HighlyPaid
        SPECIALIZES Employees
        WHERE Salary >= 60000
```

We can also create hihly–paid / good standing research assistants:

```
CREATE TABLE Student
                StName           CHAR(20),
                StAge            INTEGER,
                GPA              FLOAT)
PRIMARY KEY(StName)

CREATE TABLE HighlyPaidGoodStudentResearchAssistants(
        SPECIALIZES Employees   WHERE Salary >= 60000,
         StudentWHERE GPA >= 3.75
        JOINED CORRESPOND Employees(EmpName, EmpAge)
                         Student(StName, StAge)
        AS Name, Age, Salary, Department, GPA
        (ResearchArea            CHAR(20)
```

6. Intelligent Database Engines

The intelligent database engine implements the features of the deductive object oriented intelligent database model, Intelligent SQL. A query or a request is submitted to the intelligent database interface. The query is compiled, optimized, and executed through the different architectural components discussed here.

The components of the intelligent database engine are illustrated in Figure 3. The extended Intelligent SQL is compiled to an intelligent database virtual machine code. Many inference and database engines use a similar strategy. For instance the Warren engine has its own virtual machine instruction set which gets interpreted by the underlying system. The "instruction set" which corresponds to the functionality of the intelligent database engine is much richer than the instruction set of any existing inference or database engines. It incorporates instructions for the deductive, object–oriented, and multi–media data types supported by the engine.

The following are the fundamental components of the intelligent database engines:

The Optimizer – One of the most important modules of he engine is the optimizer. The Intelligent SQL which is submitted to the database engine is compiled and globally optimized. In other words, the optimization takes into account the whole query or program. This is in sharp contrast to the piece–meal optimization of loosely coupled database and inference engines. Global optimization greatly improves upon the performance of loosely coupled systems, which could be prohibitive for large or serious applications.

The Inference Engine – Although some of the functionality of a "traditional" inference engine is handled by other modules of the intelligent database engine, there are some specific tasks that are performed by an inference engine embedded in the intelligent database engine. The inference engine handles the flow of control of the forward or backward chaining inferencing in the execution of the programs. The inference engine also keeps track of the explanations to provide to the user the reasons and the flow of control on goals that succeed or fail. Another responsibility of the inference engine is the uncertainty algorithms. Since rules can be asserted with associated certainty factors, goals ar proven with certainties. The deduction and evaluation of certainty factors for rules is handled by the inference engine.

The Transaction Manager – The transaction manager encapsulates all the algorithms that are needed to perform concurrency control on concurrently executing transactions. Some intelligent database applications such as CAD/CAM or CASE have long duration transactions. To provide intermediate save–points in long duration transactions the transaction manager of intelligent databases supports nested transactions.

The Meta–Information Manager – The meta–information manager handles all the meta–data information associated with the persistent database. This includes schemata, the persistent class inheritance hierarchies, the user defined abstract data types, information on different indexes for different collections or sets of objects. In addition, information on data placement is also indicated to the meta–information manager. For instance the information that certain objects are clustered with other objects in the same storage extent (e.g. a disk page) is recorded in the meta–information manager. Both the inference engine and the optimizer of the

intelligent database engine interact with the meta– information manager to obtain different type of meta–data information during either optimization or inferencing.

The Storage Manager – This is one of the most important and complex modules of the intelligent database engine. The main techniques which are used to enhance the performance of databases include: indexing, clustering, and query optimization. Another strategy is caching or buffer management. The storage manager manages both the primary (RAM) and secondary storage of the databases. The indexes include single–key indexes, such as B–tree and multi–dimensional indexes. Multidimensional indexes are very useful in spatial data accesses. Intersection of regions or rectangles will use multidimensional indexes to accelerate the searches, much the same way single–key indexes are used in associative single key retrievals.

Multi–Media Manager – Intelligent database engines will handle information from many diverse sources: on–line databases, CD–ROMS, FAX cards, LANs, scanners, digitizers, optical disks, and so on. The Multi–Media Manager incorporates routines to handle peripherals such as scanners and digitizers as well as efficient management of multimedia devices such as write once optical disks, CD–ROMS and so on. For instance, write once optical disks would need special algorithms to allocate sectors in order to have more efficient storage utilization.The access and storage of objects on (from) peripherals is at a much lower physical level in the multi–media manager than the storage (buffer) manager. Thus the storage manager invokes the multi–media manager to perform its efficient storage management of the persistent databases.

As this brief description testifies the internal workings of and architecture of intelligent database engines is rather complex. Intelligent database engines provide a functionality which is a superset of:

- Object–oriented database engines
- Inference engines of expert system shells
- Traditional database engines
- Multi–media data access (e.g. full–text) engines

However, without a the tightly integrated intelligent database engines of the 1990s, the performance of the loosely coupled solutions will not be acceptable. Furthermore mapping the complex integrated products and objects onto relational database of the 1980s and attempting to support multi–media data types in these archaic environments will be a momentous task. The increase in complexity of the development of intelligent database or integrated products and objects on top of loosely coupled technologies will be inversely proportional to the elegance and ease of use of the next generation integrated environments. In other words, as the trend towards integration and sharing continues,

tightly cpupled intelligent database engines become a must. Intelligent SQL is an interface to intelligent database engines.

References

(Bancilhon et al 1987) *"FAD– a Simple and Powerful Database Language,"* F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez, Proc. of VLDB 1987.

(Cardelli and Wagner 1985) Cardelli L. and Wagner P., *"On Understanding Types, Data Abstraction, and Polymorphism"*, ACM Computing Surveys, Vol. 17, No. 4, December 1985.

(Khoshafian and Copeland 1986) *"Object Identity"*, S. Khoshafian and G. Copeland, Proc. of 1st Int. Conf. on OOPSLA, Portland, Oregon, October 1986.

(Khoshafian et all. 1990) "*Intelligent Database Engines,"* S. Khoshafian, K. Parsaye, H. Wong, to appear in Database Programming and Design, to appear July 1990.

(Khoshafian and Abnous, 1990) *Object Orientation*, S. Khoshafian and R. Abnous, John WIley Publications, to appear July 1990.

(Kuper and Vardi 1985) *"On The Expressive Power Of The Logic Data Model,"* G. M. Kuper and M. Y. Vardi, SIGMOD 1985, Austin, Texas, 180–187.

(Osborn and Heaven 1986) *"the Design of a Relationsl Database System With Abstract Data Types,"* S. L. Osborn and T. E. Heaven,ACM TODS, Vol. 11, No. 3, 1986.

(Parsaye et al. 1989) *Intelligent Databases*, K. parsaye, M. Chignell, S. Khoshafian, and H. Wong. John Wiley Publications, 1989.

(Shipman 1981) "*The Functional Data Model and the Data Language DAPLEX"*, ACM TODS, Vol. 6, No. 1, 1981.

(Stonebraker 1986) *"Inclusion of new types in relational database systems,"* M. Stonebraker, Proc. second Intl. Conf. Data Eng., LA, 1986.

Deductive Object Oriented Data Model

The Optimizer

Meta–Information Manager

Transaction Manager

Inference Engine

Storage Manager

Multi–Media Manager

Figure 3: The Architectural Components of The Intelligent Database Engine

Strawman Reference Model for Object Query Languages

José A. Blakeley Craig W. Thompson Abdallah M. Alashqur*
Texas Instruments Incorporated

Abstract

We begin by describing a *descriptive* reference model that identifies the important features that need to exist in object query languages. This reference model can be used as a basis for comparing these languages and helps provide a foundation for future standardization efforts in this area. We then describe the main features of an object query language that is based on the type system of C++, which we refer to as OQL[C++]. A query module based on OQL[C++] is being developed as part of the Zeitgeist OODB project at Texas Instruments. OQL[C++] extends the C++ programming language by providing support for managing and associatively querying sets of objects in a C++ program. OQL[C++] queries are orthogonal to persistence in the sense that they can be issued against persistent as well as transient sets of objects. The proposed OQL[C++], provides an evolutionary path from the industry's standard query language SQL to an object query language. Using ideas similar to those presented in this paper, one can propose OQL[X] where X is either an object data model, or the type system of an object-oriented programming language such as CLOS or Object Pascal.

1 Introduction

Several commercial object-oriented databases systems (OODB) and research prototypes have been developed in the past few years [4, 6, 12, 13] to meet the pressing demand from information technology made by several emerging and existing application domains. Many areas of OODBs are now mature enough to be considered for formal standardization. One of these areas is object query languages. In this paper, we provide a strawman reference model that can be used for comparing and reasoning about object query languages and suggest some areas where enough consensus could occur to make eventual standardization possible. The reference model describes the space of design choices that can be made in developing an object query language.[1]

Section 2 expresses the need for the creation of a *standard glossary* to provide a common vocabulary for implementors and users of object query languages. This level of standard promotes communication among people. Section 3 presents a strawman *object query language reference model*. In this reference model, we attempt to provide a framework that can be helpful for future standardization efforts in this area. The reference model describes a design space of characteristics and defines the criteria and features that serve as a basis for comparing different existing and future object query languages. Section 4 describes our object query language OQL[C++]. Section 5 provides a comparison between a representative sample of query languages based on the established reference model. Some conclusions are presented in Section 6.

2 Need for a Glossary

The purpose of a glossary is to register terms and how they are used in different object query languages. The value of a glossary in standardization is to provide a common vocabulary so we all understand common terms the same way and can distinguish their various overloaded meanings. In addition, glossary terms are important in the development of a reference model (Section 3) and provide a simple approximate way to scope a domain. In the appendix, we provide a list of important terms used in this area. Although the list is not exhaustive, it contains many of the terms that are widely used in the context of object query languages.

3 Descriptive Reference Model

A *descriptive reference model* for an object query language is an English description of the design

*Authors' addresses: Texas Instruments Incorporated, P.O. Box 655474, MS 238, Dallas, Texas 75265. E-mail: {blakeley,thompson,alashqur}@csc.ti.com.

[1] comments for the improvement of the descriptive model or OQL are solicited.

space of features that "cover" existing (and future) object query languages and provide *people* with a way to compare them.

An object query language contains operators and provides language constructs for manipulating sets of objects modeled by an object data model. At a high-level of abstraction, a query written in an object query language can be viewed as a function that, when applied to the object base (of sets of transient and/or persistent objects), returns one or more sets of objects that satisfy the predicates specified in the query. These predicates can be based on the state and/or behavior of objects, which are determined by their classes.

Different query languages have different syntax and support different techniques for expressing queries. They also vary in the sophistication of the predicates that can be specified (e.g., some of them allow a nested query to be used as an argument to a predicate). Other languages provide syntactic short cuts for expressing frequently used expressions such as using the class name as a range variable (e.g., as in GEM [24]) where there is no ambiguity. Query languages also differ in their support for data abstraction where some of them restrict queries to be based on behavior (methods) such as HP Iris' OSQL [8], while others such as MCC ORION [7] allow queries based on state (in order to simplify the query optimizer). In a design space of multiple design choices, it may be hard to decide on an "absolute best" combination of design alternatives. However, there appears to be a general but not unanimous consensus regarding certain design choices. We refer to these design choices as the common features of object query languages and they are described in subsection 3.1. Other design choices where consensus seems to be further away are termed controversial features and are described in subsection 3.2.

3.1 Common Features

The following can be considered to be *common features* of object query languages. Each of these features represents a preferred desgin choice between two possible alternatives in the design space of object query languages. They are considered "common" because people have studied them in detail and have experimented with them in several prototype systems. They can be used as a starting point for setting up standards in the area of object query languages.

SELECT-FROM-WHERE paradigm. One of the choices that needs to be considered at an early stage of the design process is whether or not to make use of the SQL heritage in the object query language to be developed. Since SQL is a standard [1], it is advantageous from an industrial perspective to evolve from it toward an object query language rather than to introduce a completely new syntax. Several query language designers have chosen to cast the syntax of their languages into the Select-From-Where (SFW) structure to maintain an upward compatibility with SQL and to smooth the migration to the new languages. The semantics as well as the expressions that can be specified in the Select, From, and Where clauses of an object query language are however different from those in SQL in that these expressions are richer in object query languages (e.g., path expressions, behaviour) to be able to exploit the richness and sophistication of the object data model. A possible disadvantage of this approach is that the familiarity with the syntax and semantics of SQL by many practitioners may hinder understanding the new semantics associated with the SFW paradigm of SQL, especially since the shift from the old semantics to the new semantics is relatively huge.

Path expressions. Many of the existing object query languages allow for logical navigation at the schema level by specifying path expressions. For example, if Class1 has an attribute whose type is Class2 and Class2 has an attribute whose type is Class3 in some schema, then Class1.Class2.Class3 is a path expression that starts at Class1 and ends at Class3. Path expressions can be used in specifying predicates or identifying the list of attributes to be retrieved. For example, "Class1.Class2.Class3 == value" is an associative predicate that identifies all the Class1 objects whose related Class3 objects are equal to the given value. Different query languages may use different syntax to express path expressions. For example, the above path expression is expressed by functional query languages as "Class3(Class2(Class1))." Supporting path expressions in an object query language does not violate the principle of physical data independence since the query optimizer may independently choose the appropriate access paths at the physical level in order to evaluate the given query. Languages that respect data abstraction permit only the use of messages in expressing path expressions. This is discussed in the following subsection in more detail. Path expressions are used for querying complex objects by

specifying predicates on attributes that are deeply nested within the structure of these objects.

Inheritance. Almost all of the existing object query languages make use of the inheritance property of the object data model. Objects of a class can be uniformly queried based on the attributes defined in their class as well as those inherited from its super classes. However, not all object query languages support multiple inheritance and those that support it follow different conflict resolution strategies to resolve name ambiguities.

Explicit joins. In the expression Class1.Class2 there is an *implicit* (entity-based) join between objects belonging to Class1 and Class2. Clearly, support for *explicit* (value-based) joins similar to those of the relational query languages is needed. This permits specifying new relationships among objects not captured by their class definitions, based on type-compatible values.

Set-valued attributes. In addition to supporting scalar-valued attributes and providing scalar comparison operators for specifying predicates, many object query languages also support set-valued attributes. Set comparison operators are provided in these languages to enable specifying predicates on such attributes. Also, several of these languages support the use of existential and universal quantifiers on sets.

Nested queries (subqueries). The concept of a *nested query* is well known and is supported by relational query languages. Several object query languages also support this feature. Query languages that support set comparison predicates allow the use of nested queries as arguments to these predicates.

Aggregate Functions. Several object query languages support the use of aggregate functions (e.g., COUNT, AVERAGE, MAX) in specifying queries. Again, aggregate functions can be used in the formulation of predicates on which retrieval is to be performed and/or in the target list specification (i.e., in a Select clause).

3.2 Controversial Features

In this section, we describe *controversial features* of object query languages. Each of these features is supported by only a few of the existing query languages and further research and experimentation is needed before there is an acceptable general consensus regarding each feature. These features can be tackled by the standards community in a later stage after considering the more common features described in the previous section.

Data abstraction. Some object query languages restrict the access to objects to be through their operational (behavioral) interface. The only way an object can return a value is by responding to a message that triggers a method that computes or retrieves the needed value. In this approach, functions can be used in the Where clause to define predicates and in the Select clause to retrieve related objects. Other query languages allow queries to be formulated in terms of the object's state in favor of simpler query optimizers. Data abstraction is important becuase it provides data independence, thus object query languages that support this feature achieve a higher degree of data independence.

Functions with or without side effects. Several object query languages that support data abstraction allow only side-effect-free functions to be used in the specification of queries. Functions with side effects may create undesirable changes to the state of the database that may not be easily detected. This complicates the task of the database integrity manager. Also, a decision needs to be made about the language that can be used to write these functions or methods.

Queries that result in derived relationships between objects. Some object query languages allow queries to compute a join between objects and return that as a result. In other words, the result of a query is not a single set of objects but two (or more) sets of objects with derived relationships between objects of the two sets. Some other query languages support this feature by allowing queries to return new types whose instances are tuples of OIDs. Further research is needed in the area of derived behaviour.

Null values. Few object query languages support the manipulation of *null* values. For example, computing the logical AND between two predicates if one or both of them returns a null value is not possible in many of these languages. In addition, the notion of returning null values as part of the result of a query (along the lines of the concept of "outer join" in the relational world) has not been adequately addressed in the world of object query languages.

Recursion. Commercial relational query languages do not support recursion because of the difficulty in providing efficient implementation techniques. Database researchers have identified several useful notions of recursion. Among these, *traversal recursion* [17] seems to have a direct application in object query languages because of the direct map-

ping of a complex object into a graph. Most existing object query languages have not yet adequately addressed this issue.

Relationship to the programming language's type system. One of the design choices that can be identified early in the design space is whether to design the query language based on an object data model that is independent of the type system of any programming language or to design it for a specific programming language's type system. In the first approach, a single object query language that is independent of the syntax and type system of any programming language needs to be embedded in these programming languages. The advantage of this approach is that the query language syntax is uniform across different programming languages, which reduces the learning efforts by programmers who frequently switch among languages. Also, one language-independent persistent storage can be used across all applications. In the second approach, the query language is customized (or designed) to suit the type system and the syntax of a given programming language. This approach yields a spectrum of object query languages OQL[X], where X corresponds to the type system of a general purpose programming language (e.g., C++, CLOS) to which the query language is customized. The advantage of this approach is that it produces a better integration between the query and programming languages.

Orthogonal treatment of types and type extents. Orthogonal treatment of types and type extents may need to be considered in the query language. Not every object must necessarily exist in a set. Objects not in any set can only be manipulated by the constructs of the programming language in a navigational object-at-a-time manner while objects in sets can in addition be associatively queried. This requires that a user needs to explicitly create and populate sets in his program.

Seamlessness. When a query facility is integrated with a programming language, it is desirable from a programmer's point of view to be able to manipulate sets of transient as well as persistent objects in a uniform (seamless) way. In addition, retrieving objects form the persistent storage to the program space should be as transparent as possible to the programmer.

Strong typing. Strong typing in a programming language guarantees that the arguments of functions are type-correct; a programming language with strong typing is viewed as safer and more efficient than one that does not support enforcement of types. In current SQL embeddings, the representation of the host language and the database query language are different and the programmer must break strong typing to copy his data to the database. Experimental persistent programming languages correct this problem, but they lack query extensions. Many of today's query languages do not maintain type-safety at this interface, which is a desirable feature to be incorporated in future systems.

Result generation and result presentation. Unlike a relational query which returns a set of tuples of values that are readable by the user, an object query may return set(s) of OIDs. A set of OIDs can be further manipulated (e.g., as arguments to set comparison predicates) or used by other functions but it cannot be presented to the user since OIDs are system-generated identifiers that do not carry any meaning from the user's point of view. A presentation manager is to take the set of OIDs returned by a query and presenting their related data to the user (in the form of a table, a histogram, etc.). The presentation manager may use Nest and UnNest operations or a hypermedia interface to produce the desired format. Most existing object query languages and systems mix the functions of result generation and the result presentation.

Table 1 in Section 5 presents a comparison among a representative sample of query languages (including OQL[C++] which is described in the next section) in terms of the common and controversial features they support.

4 The Object Query Language OQL[C++]

In this section, we describe the syntax and semantics of the object query language OQL[C++].[2] Current database management systems (DBMS) allow query languages (e.g., SQL [10], QUEL [21]) to be embedded in programming languages (e.g., C, Fortran, Cobol). The difference between the type systems of the programming language and the embed-

[2] OQL[C++] is a module of the Zeitgeist Open OODB. The current implementation of the C++ version of Zeitgeist already provides Persistent C++ (PC++) [16] as an extension of C++. PC++ allows objects to persist between different programming sessions. PC++ uses a preprocessor that translates PC++ programs into C++ programs that can be compiled by off-the-shelf compilers. The OQL[C++] module of Zeitgeist extends the capabilities of PC++ by providing support for maintaining and associatively querying sets of objects.

ded query language is highly visible to the application programmer. This results in the following problems and limitations. First, strong typing is lost at the database interface and it is a responsibility of the programmer to perform the mapping from database to application program data structures. By leaving this translation up to application programmers, the type-safe, strong-typing is potentially broken (not guaranteed by the system) exactly at the program-database interface. this loss of strong typing is a big problem because, within programs, type-safe behavior is viewed as a major feature and the data stored in the database is viewed as an enterprise's chief resource. Second, set-oriented queries can only be formulated to retrieve persistent database objects (e.g., relations) to the program work space; they cannot be formulated against transient objects nor against persistent objects after they enter the program space. Third, user defined functions (written in the programming language) cannot be used in queries. Similarly, queries cannot be freely mixed with programming language statements (e.g., as parameters to functions) and they need to obey specific protocols. Fourth, programming languages typically do not provide SET type as one of their basic types, consequently, sets of objects returned by a query cannot be manipulated directly by the programming language constructs. This forces the introduction of artificial extensions to the language (e.g., SQL cursors) to bind database objects to programming language variables so that queries return their results to the program one object at a time. Finally, the syntax and semantics of the query and programming languages are completely different and the programmer has to learn and be aware of these differences. Consequences are that only a small percentage of application programmers know how to use embedded SQL, and that the advantage of high-level querying (set-oriented specification of a database copy) is not available to the programmer.

The term "OQL" refers to the specific method of extending any programming language with associative query statements, while "OQL[C++]" refers to the particular coupling of OQL with C++ which allows certain C++ expressions to be used in the formulation of queries. In general, the term "OQL[X]" could be used to designate a coupling of OQL with a programming language X, where X is, for example, C++, CLOS, Smalltalk, or Objective-C.

Basically, OQL adopts the SFW structure of the SQL Select statement and allows appropriate statements of the programming language to be combined within queries. The SFW structure of SQL is adopted because it provides a standard model for the formulation of queries in object-oriented programming languages that currently enjoys wide use in database applications.

OQL[C++] extends the type system of C++ with the support of a parameterized set type, thereby enabling C++ to handle sets. Also, OQL[C++] is more than an embedding of a query language in C++ because it provides a better integration between query and programming language statements by allowing the sets returned from an OQL[C++] query to be used in any place within a C++ program expecting to handle a set (e.g., a parameter to a C++ function), allowing set-valued C++ functions to be used in the From clause of an OQL[C++] statement, and allowing Boolean-valued C++ functions to be freely combined as part of a predicate in the Where clause of an OQL[C++] statement.

Unlike many of the existing object-oriented query languages [7, 8, 9, 11, 14], OQL[C++] provides a better integration with the object-oriented programming language, C++. The design objectives achieved by OQL[C++] are described below.

Minimal data model dependency. This is achieved by assuming only the knowledge of the type system of the programming language (in this case C++) as a "bare-bones" data model rather than defining a new proprietary data model from scratch. In our approach, the types and concepts supported by a programming language (e.g., object, class, data member, member function, inheritance) provide the basis of an object data model on which the query language operates.

Data abstraction. OQL[C++] requires all queries to be formulated in terms of the object's public interface (member functions), thus hiding the object's internal representation. This makes the principle of *data independence* (i.e., the immunity of applications to changes in the data structures used to organize data), which is one of the major advantages introduced by the relational model, to be preserved in object-oriented database queries.

Explicit, user-maintained sets. OQL[C++] requires programmers to define variables of type OQL_SET in their programs. This makes the concepts of class definition and class extent (the set of instances of a class) to be orthogonal. As a result of this, not all classes may have sets associated with them and some classes may more tha␣ one set associated with them, unlike database systems which

support implicit sets where a class (schema) always has one set associated with it.

Queries on transient and persistent sets. In relational DBMSs and in all current OODBMSs, queries are performed exclusively on persistent data. In OQL[C++], it is possible to define and query transient sets, that is, sets whose life span is a single program invocation. This makes the OQL[C++] approach to querying independent of whether data is maintained persistently in a DBMS or transiently in the application's workspace.[3] As a direct consequence of this independence, OQL[C++] can be used as a tool for supporting query statements in a programming language.

Better integration with the programming language. OQL[C++] allows a more flexible combination of query and programming language statements. This is achieved by allowing typed range variables and user-defined, set-valued functions in addition to OQL_SETs in the From clause; user-defined, Boolean-valued functions and inherited member functions in the Where clause; and typed objects and substitutability in the Select clause of a query. This enables OQL[C++] to be much more programmable than SQL without requiring the special purpose, yet-another-language extensions to SQL to extend its DML functionality. Also, the uniformity of the type systems of the programming and query languages achieved in OQL[C++] retains strong typing at the query interface, making database programming much easier and more reliable than conventional databases.

Additionally, OQL[C++] provides elegant *looping* constructs for C++. The programmer can take advantage of SQL's relational calculus to specify *what* data needs to be retrieved and let the query processor map the higher level specification into loops that determine *how* to retrieve the data efficiently. This increases the programmer's productivity by relieving him/her from the burden of determining efficient access paths and by making his/her code easier to write and to read.

We believe that OQL[C++] provides a reasonable approach to querying from within a C++ program by combining essential features of C++ with the industry's standard query language SQL. We also believe OQL represents a reasonable evolutionary path from SQL to "Object SQL."

The following subsection describes the C++ data model for which OQL[C++] has been designed. OQL[C++] itself is described in Section 4.3. Finally, an example of OQL[C++] queries used as part of C++ programs is given in Section 4.4.

[3] In fact, the orthogonality to persistence of OQL sets is a byproduct of the fact that OQL[C++] is coupled with Persistent C++ which provides orthogonality of types with respect to persistence.

4.1 The C++ Data Model

In this section, we briefly describe the data model (also, referred to as the *type system*) of the C++ object-oriented programming language, used as a basis for OQL[C++] [22].

The C++ data model includes a set of predefined (built-in or primitive) data types: character, integer, long, short, float, and double. They can be used in defining more complex user defined types by means of arrays, structs, or classes.

A class is a user-defined type that determines the *structure* and *behavior* of a collection of objects. The definition of a class involves defining a set of *data members* (attributes or instance variables) and a set of *member functions* (operations or methods). A *class member* (a member function or a data member) can be declared as public, private, or protected. A public member can be accessed from anywhere in a program. A private member can be accessed only by the member functions of its class. A protected member can be accessed by the member functions of its class as well as the member functions of the classes derived from it (derived classes are described later in this section). An *abstract data type* is a class that has a public set of member functions and no public data members (all its data members are declared as private or protected).

When defining a class, the type of each of its data members needs to be specified. The type of a data member can be one of the predefined data types (e.g., integer, float) or it can be another user-defined type. This capability in C++ allows for building aggregation (some times referred to as composition) hierarchies or graphs to represent complex objects, which are recursively defined in terms of other objects. Recursive definitions in which the type of a data member of a class is the same class are also possible. A class can be derived from one or more base classes (thus, multiple inheritance is supported). A derived class can itself serve as the base class for other derivations. This capability in C++ allows for building generalization hierarchies or lattices. A derived class is referred to as a subtype of its base class. A derived class inherits all the public and protected members of its base class. Private members of a base class are not accessible from any other class including derived classes.

Finally, C++, like many programming languages supports free composition of classes, constructors, and primitive data items. But, like many programming languages, C++ does not support sets as data constructors. OODB systems, on the other hand, support sets as type extents either *implicitly* or *explicitly*. Examples of implicit support of sets can be found in systems like Iris [12], ODE [4], and Orion [6]. In these systems, for each class (type) defined, the system creates a set in which all the objects of the class are stored. Queries can then be targeted to these sets. In other systems (e.g., EXODUS [9] and Encore [18]), the user has to explicitly create those sets. In this case, multiple sets corresponding to a class can be created where each set may contain a subset of all the instances of the class. This is useful in applications where certain persistent objects of some class will be accessed by queries. Objects of a class may be grouped into different subsets based on some shared semantics (i.e., without having to create corresponding subtypes). Consider a class `Patient` defining information about patients in a hospital. In addition to the set of all patients in the hospital it may be convenient to allow the declaration of special sets of patients such as laboratory patients or pediatrics patients.

In OQL[C++], sets are declared explicitly by the user in a C++ program. OQL[C++] sets are instances of C++ parameterized classes[4] declared and defined using the statements

```
DECLARE OQL_SET <class-name>;
IMPLEMENT OQL_SET <class-name>;
```

respectively. `DECLARE` and `IMPLEMENT` are keywords recognized by an OQL[C++] preprocessor. They trigger the generation of necessary C++ code to declare and define the parameterized `OQL_SET` class, where `OQL_SET` is the name to be given by the user to the created set (e.g., Lab_Patients). The parameter `class-name` is the name of a previously declared class (e.g. Patient) which represents the type of the instances permitted in the set. The public member functions of the `OQL_SET` class include functions to: (a) establish membership in the set (e.g., `add`, `delete`, `find`), (b) perform set operations (e.g., `intersection`, `union`, `difference`), (c) iterate over the members of the set, and (d) create and drop indices on the set based on values returned by public member functions of the parameter class.[5] Set management in OQL[C++] is orthogonal to persistence, thus it is possible to have transient sets of persistent and/or transient objects, or persistent sets of persistent objects all of which can be queried.

4.2 The Query Language OQL[C++]

In this section we present the object query language. The description is illustrated by queries against the schema of Figure 1 which represents clinical information about patients. Names in the graph represent classes. Composition (aggregation) and inheritance among classes are represented by continuous and dashed lines, respectively. For simplicity, data members and member functions are not shown in the figure.

A query block in OQL[C++] is similar to that of SQL and is represented as follows:

```
SELECT <objects>
FROM <range variable> IN <set>
WHERE <predicate>;
```

The Select clause identifies the type of the objects in the answer set. Range variables are declared in the From clause. Several variables ranging on several sets may be declared in this clause. The Where clause specifies the predicate that defines the properties to be satisfied by the objects to be retrieved. In the following examples, we assume that there is a public member function corresponding to each data member of a class that returns the data member value(s). For example, `get_physician()` is a public member function of `Patient`, which when invoked by a particular patient object, it returns the related physician object. The bidirectional arrow between Patient and Physician indicates that Patient has a class member of type Physician and that Physician has a class member of type Patient. Arrows in the above diagram represent member functions. The following are some examples.

Example 1. Retrieve the patients who are treated by Dr. J. Smith.

```
SELECT p FROM Patient p IN Patient_Set
WHERE p.get_physician().name() == ''J. Smith''
```

In this example, p is declared in the From clause as a range variable over the instances of Patient_Set.

[4] Declaration and definition of OQL_SET classes in OQL[C++] are identical to the way parameterized sets are declared and defined in Texas Instruments' C++ Object Oriented Library (COOL) [3] which provides an implementation of parameterized types as proposed by Stroustrup [23].

[5] Implicit (system-maintained) sets can also be declared for classes by adding a keyword IMPLICIT-SET to the class declaration.

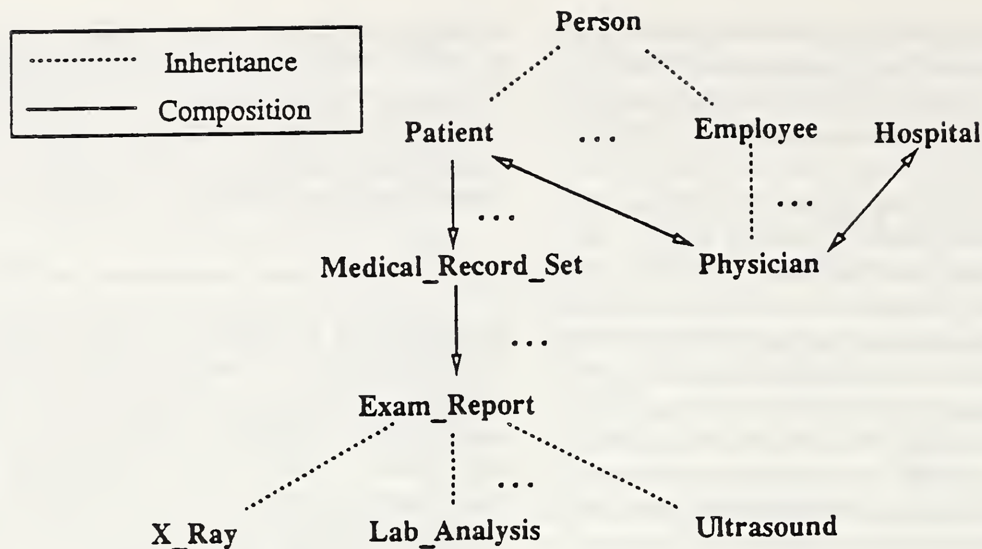


Figure 1: A hospital database.

(Assuming that Patient_Set has been declared in a C++ program using the DECLARE statement described in Section 2 and contains a set of instances of type Patient). The member name referenced in the Where clause is a public member function of Person inherited by Patient and Physician. The Select clause states that the objects returned by the query belong to the class Patient. In other words, the objects in the answer set of the query are Patient objects that behave exactly as Patient objects defined in a C++ program. *Path expressions* (e.g., p.get_physician().name()) allow the formulation of predicates on values deeply nested in the structure of an object. There may be *single-valued* or *set-valued* path expressions. Example 3 shows how set-valued path expressions are handled in OQL[C++].

In OQL[C++] it is also possible to declare the type of the members of the answer set to be any of the supertypes of Patient. This type conversion can be specified in the Select clause and is consistent with the concept of substitutability in C++. For example, the following query is directed against Patient_Set and it returns a set containing Person objects by casting the returned Patient objects into Person objects. Therefore, only the member functions of Person in this case can be applied to the instances of the answer set.

```
SELECT (Person) p
FROM Patient p IN Patient_Set
WHERE p.get_physician().name() == ''J. Smith''
```

The assignment operator can be used to save the answer set and give it a name. For example, the answer set in the above query can be named as Smith_Patients and saved for later use by other queries where Smith_Patients is a C++ program variable of type OQL_SET <Patient>. For simplicity, the rest of the examples omit the assignment to a result variable.

```
Smith_Patients = SELECT p
  FROM Patient p IN Patient_Set
  WHERE p.get_physician().name() == ''J. Smith''
```

Example 2. Retrieve the patients who are treated by Dr. J. Smith and who are more than 60 years old.

```
SELECT p
FROM Patient p IN Smith_Patients
WHERE p.age() > 60
```

Where Smith_Patients is the set returned by the previous query.

Example 3. Retrieve male patients who have been diagnosed with flu prior to October 10, 1989. (Medical_record is a set-valued attribute of Patient.)

```
SELECT p
FROM Patient p IN Patient_Set
WHERE p.sex == ''male'' &&
  EXISTS ( SELECT r
    FROM Medical_record r IN
         p.get_medical_record()
    WHERE r.get_date() < ''10/10/89'' &&
      r.get_diagnosis() == ''flu'' );
```

Because the path expression p.get_medical_record() is set-valued (i.e., it returns a set of medical records), it is necessary to define a variable r to range over the members of that set. This leads to the use of nested queries (subqueries) in an OQL[C++] statement. The EXISTS keyword in OQL[C++] maintains the same semantics as in SQL.

Predicates in the Where clause can be defined using comparison operators $\theta \in \{==, <, <=, >, >=, !=\}$, and logical operators && (AND), || (OR), and NOT. Note that the syntax of comparison operators is the same as in C++. Valid *atomic terms* are: $t_1\ \theta\ t_2$, $t_1\ \theta\ c$, t_1 IN s_1, s_1 CONTAINS s_2, $v\ \theta$ ALL s_1, $v\ \theta$ ANY s_1, and EXISTS s_1; where t_1 and t_2 are single-valued path expressions, s_1 and s_2 are sets, v is a single-valued path expression or a constant, c is a constant (integer or string), and θ is a comparison operator. The atomic terms involving ANY and ALL are used for existential and universal quantification, respectively. A *predicate* is a Boolean combination of atomic terms.

Data abstraction represents one of the main advantages offered by OQL[C++]. Consider the bidirectional relationship between Hospital and Physician shown in Figure 1. Assume that the Hospital → Physician link is represented explicitly in the Hospital class by a physician-valued member function get_physician(), and that the Physician → Hospital link is represented implicitly in the Physician class by a get_hospital() function (note that no data member in Physician maintains this information) defined as:

```
Hospital get_hospital() {
  return( SELECT h FROM Hospital IN Hospital_Set
          WHERE h->get_physician() == this );
}
```

Data abstraction allows OQL[C++] queries on composite objects to be formulated using a uniform mechanism (i.e., path expressions) without regard to the way composition is implemented. This is illustrated by the following example.

Example 4. Retrieve patients whose ages are less than 19 years old and who are treated by physicians who work for hospitals located in Dallas.

```
SELECT p
FROM Patient p IN Patient_Set
WHERE p.age() < 19 &&
  p.get_physician().get_hospital().location()
    == "Dallas''
```

Other query languages that do not support data abstraction require different formulations for queries on composite objects depending on the way the composite relationships are implemented. In these languages, an explicit join with objects returned by a nested query is necessary to express the above query as shown below.

```
SELECT p
FROM Patient p IN Patient-Set
WHERE p.age() < 19 &&
      p.physician IN (SELECT s.Physician
  FROM Hospital s IN Hospital_Set
  WHERE s.location() == "Dallas'')
```

Example 5. Retrieve patients having X-ray exams matching a tuberculosis of the lungs pattern.

```
X_Ray_List *f( Patient * p ) {
  X_Ray_List * x =
    p->medical_record()->exam_list.extract(X_RAY);
  return( x );
}

X_Ray_Set * Make_set( X_Ray_List * l ) {
  X_Ray_Set * x;

  x = new X_Ray_Set;
  l.reset();
  for(X_Ray *p = l.value(); !(l.end()); l.next()){
     x.add( p );
  };
  return( x );
}

SELECT p
FROM Patient *p IN Patient_Set
WHERE EXISTS ( SELECT *
  FROM X_Ray_Set *r IN Make_set(f( p ))
  WHERE x_ray_match(r->picture(), Bitmap *pattern)
          )
```

This query illustrates the use of set-valued, user-defined functions in an OQL[C++] statement. Assume, every medical record of a patient contains a heterogeneous list of laboratory exams (e.g., X rays, ultrasounds, blood tests). To be able to query on a set of X rays, it is necessary to first extract the X ray objects from the list of laboratory exams (see function f above) and then make the list into a set (see function Make_set above). This is necessary because OQL[C++] can only query sets of homogeneous objects. The user-defined (Boolean) function x_ray_match compares the bit pattern r->picture() of an X ray with program

variable of type Bitmap holding a typical tuberculosis pattern.

4.3 Queries in C++ Programs

This section illustrates the use of OQL[C++] within C++ programs. OQL[C++] provides a much better coupling of a query language with a programming language than previous embedding language approaches as illustrated by the example program of Figure 2.

In the example, Statements 2 and 3 show the way to declare and define a parameterized set of patients. Statement 6 declares two program variables `mypatients`, and `result` of type `OQL_SET<Patient>`. Statement 8 shows how sets are populated using the add member function. Statements 10–12 show a way to iterate through all the individual members of a set. Statement 14 shows the query of Example 5 on the set `mypatients`. Note the use of user-defined functions as part of the program. Statements 16-19 represents code that defines the functions `f` and `Make_set`.

OQL[C++] queries are recognized by a preprocessor which parses, optimizes, and translates them into efficient C++ or Persistent C++ code (depending on whether the set is transient or persistent). The resulting program is then compiled by a standard C++ compiler. Until incremental C++ compilers/interpreters are available, OQL[C++] will not provide interactive queries.

5 Comparison Among Query Languages

In this section we compare some object query languages based on some of the common and controversial features described in Section 3 of this paper. The languages included in this comparison are OSQL [8], ORION query language [7], the object-oriented query language developed at the Universiy of Florida OQL(UF) [5], EXCESS [9], GEM [24], Query Algebra [19], O++ [4], Postquel [20], OPAL [2], RELOOP [11]. Object Design Query language [15], and OQL[C++].

In the above table, N stands for No, Y stands for Yes, and ? stands for unknown or unclear. Rows in the table describe languages and columns describe features.[6] The first seven columns are for the common features while the others describe the controverial features. These features are numbered in the table as follows.

1. SQL SELECT-FROM-WHERE paradigm
2. Path expressions
3. Support for inheritance
4. Explicit joins
5. Set-valued attributes
6. Nested queries
7. Supporting aggregate funcitons
8. Respecting data abstraction
9. Support for using functions in queries
10. Support for queries that result in join objects
11. Support for Null values
12. Recursion
13. Makes use fo the type system of a programming language
14. Integration with a programming language has been considered
15. Orthogonal treatment of types and type extents
16. Orthogonal treatment of result generation and result presentation

[6] Although we have tried to make our comparison as precise as possible based on published material, we know the table may still not be accurate.

6 Conclusions

This paper has proposed a reference model for comparing object query languages, identified areas where consensus is possible and standards can emerge, and areas where further experimentation is needed before consensus can be reached.

We have proposed an object query language OQL[C++] which provides a better integration of query and programming languages by adopting the SFW structure of SQL and extending it with C++ expressions. The language supports strong typing, data abstraction, queries on transient and persistent sets, user-maintained class extents, and a better combination of query and programming language statements.

We feel that work on standards for object query languages needs to be actively pursued by standards committees in parallel with the current efforts in other aspects of object-orientation. Such committees include the X3/SPARC/DBSSG/OODBTG on a reference model for object databases, X3H2 on SQL3, Object Management Group (OMG) on an object software framework, X3J13.1 on CLOS, and X3J16 on C++. We propose the creation of a X3 subcommittee formed from members of the two committees X3H2 (SQL) and X3J16 (C++) to work on a proposal for a standard on OQL[C++].

```
1.  #include        <OQL_Set.h>          // header file containing templates
2.  DECLARE         OQL_Set<Patient>     // declares a set
3.  IMPLEMENT       OQL_Set<Patient>     // defines functions of a set
    ...
4.  main()
5.  {
6.    OQL_Set<Patient> mypatients, result;
7.    Patient p1, p2;
      ...                                // code that creates instances of
                                         // of patients not shown
8.    mypatients.add( p1 );              // add a member to the set
9.    ...
10.   mypatients.reset();                // set iteration
11.   for( Boolean t=mypatients.next(); t!=INVALID; t=mypatients.next() ) {
12.     p2 = mypatients.value();
        ...
13.   };
      ...
14.   result = SELECT p FROM Patient *p IN mypatients    // query of Ex. 6
              WHERE EXISTS (
                    SELECT * FROM X_Ray_Set *r IN Make_set( f( p ) )
                    WHERE x_ray_match( r->picture(), Bitmap *pattern )
                    )
15. };
16. X_Ray_List *f( Patient * p ) {
    ...
17. }
18. X_Ray_Set * Make_set( X_Ray_List * l )
{   ...
19. }
```

Figure 2: Example using OQL in a C++ program.

An important byproduct of the work presented in this paper is that it offers a way to simplify extensions being considered for SQL. In particular, today's proposals on SQL3 range from how to add abstract data types (ADTs) and inheritance to how to extend the expressiveness of the query language toward computational completeness. By combining SQL with a programming language's type system "X" to yield SQL[X], the essential value of set-oriented queries is preserved (in fact, added to the programming language), the interface between the programming language and the database preserves strong typing, and the programming language supplies a computationally complete data manipulation language and a tested data definition language.

Appendix

A Proposed terms for a glossary

database language (i.e., DDL & DML)
host programming language, general
  purpose programming language
database programming language
embedded language, impedance mismatch,
  seamlessness (w.r.t. a data model
  or language)
query language, calculus, relational algebra,
  object algebra, graph algebra
logical database language, physical database
  language
relational completeness, closure property,
  computational completeness

query, query block, nested query, subquery
set, collection, bag
set-oriented query, associative query

Language	Common Features							Controversial Features								
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
OSQL	Y	Y	Y	Y	Y	Y	?	Y	Y	?	N	N	N	Y	?	N
ORION	N	Y	Y	Y	Y	N	N	N	Y	Y	N	Y	N	N	N	Y
OQL (UF)	N	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	N	N	N	Y
EXCESS	N	Y	Y	Y	Y	Y	Y	N	Y	?	N	N	N	?	Y	?
GEM	N	Y	Y	Y	Y	?	Y	N	N	N	Y	N	N	N	N	N
Query Algebra	N	Y	?	Y	Y	Y	N	N	?	Y	?	N	N	N	Y	Y
O++	N	?	Y	Y	Y	N	N	N	?	N	N	N	Y	Y	N	N
Postquel	N	Y	Y	Y	N	Y	Y	N	Y	Y	N	N	N	N	N	N
OPAL	N	Y	Y	N	?	?	N	N	Y	N	N	N	Y	?	Y	N
RELOOP	Y	Y	?	Y	Y	Y	Y	Y	?	?	N	N	?	?	N	?
Object Design	N	Y	Y	?	?	?	N	N	?	N	N	N	Y	Y	?	?
OQL[C++]	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	N	N	Y	Y	Y	Y

Table 1: Comparison between a representitave sample of query languages.

object-at-a-time retrieval, navigation
view definition
transaction

existential quantifier
universal quantifier
target list, target set
range variable
aggregate function
path expression
predicate, scalar comparison operator, set comparison operator
nest, unnest, desset, flatten
join, projection, selection
graph operator
set iteration, iterator
set operators: union, intersection, difference
recursion, linear recursion, traversal recursion

data model, object data model, type system
object database, object base
object identity, object identifier, object, entity
shallow equality, deep equality
inheritance, generalization, specialization
class (type), subclass (derived class), superclass, metaclass
composition, aggregation, complex object
extension, extent, intension
extensional database, intensional database
explicit type extent, explicit set creation
implicit type extent, implicit set creation

behavior, function, member function, method, message, message passing
attribute, relationship, link, property, data member, instance variable, slot, state, representation
scalar-valued attribute, set-valued attribute
encapsulation, data abstraction, abstract data type

integrity constraint, referential integrity, relationship, mapping constraint, cardinality constraint
strong typing
set inclusion semantics

persistent language
persistent object, persistent class, persistent set
transient (volatile) object, transient class, transient set

query processing, query translation, query optimization, execution plan
query graph, query tree, operator tree
index

References

[1] American National Standard for Information Systems - database language – SQL. Tech. Rep. ANSI-X3.135-1986, American National Standards Institute, Inc., 1430 Broadway, New York, NY 10018, Oct. 1986.

[2] Programming in OPAL. Tech. rep., Servio Logic Development Corp., Beaverton, Oregon, 1986.

[3] C++ Object Oriented Library User's Manual. Tech. rep., Texas Instruments Incorporated, Information Technology Group, P.O. Box 149149, MS 2151, Austin, Texas 78714-9149, Mar. 1990.

[4] *Agrawal, R., and Gehani, N. H.* ODE (Object Database and Environment): The language and the data model. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data* (Portland, OR, May 1989), pp. 36–45.

[5] *Alashqur, A., Su, S., and Lam, H.* A Rule-Based Language for Deductive Object-Oriented Databases. In *Proceedings of the Sixth International Conference on Data Engineering, Los Angeles* (1990), pp. 58–67.

[6] *Attwood, T., and Orenstein, J.* Notes Toward a Standard Object Oriented DDL and DML. In *Proceedings of the First OODB Standardization Workshop* (Atlantic City, NJ, 1990).

[7] *Banerjee, J., Chou, H.-T., Garza, J. F., Kim, W., Woelk, D., Ballou, N., and Kim, H.-J.* Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems 5*, 1 (Jan. 1987), 3–26.

[8] *Banerjee, J., Kim, W., and Kim, K.-C.* Queries in Object-Oriented Databases. In *Proceedings of the Fourth International Conference on Data Engineering, Los Angeles* (Feb. 1988), pp. 31–38.

[9] *Beech, D.* A Foundation for Evolution from Relational to object databases. In *Advances in Database Technology - EDTB '88*, vol. 303. Springer-Verlag, 1988, pp. 251–270.

[10] *Carey, M. J., DeWitt, D. J., and Vandenberg, S. L.* A Data Model and Query Language for EXODUS. In *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data* (Chicago, IL, 1988), pp. 413–423.

[11] *Chamberlin, D., and et.al.* SEQUEL 2: a Unified Approach to Data Definition, Manipulation and Control. *IBM Journal of Research and Development* (Nov. 1976).

[12] *Cluet, S., Delobel, C., Lécluse, C., and Richard, P.* RELOOP, and Algebra Based Query Language for an Object-Oriented Database System. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases* (Dec. 1989).

[13] *Fishman, D. H., Beech, D., Cate, H. P., Chow, E. C., Connors, T., Davis, J. W., Derrett, N., Hoch, C. G., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M. A., Ryan, T. A., and Shan, M.* IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems 5*, 1 (Jan. 1987), 48–69.

[14] *Ford, S., Joseph, J., Langworthy, D. E., Lively, D. F., Pathak, G., Perez, E. R., Peterson, R. W., m. Sparacin, D., Thatte, S. M., Wells, D. L., and Agarwala, S.* ZEITGEIST: Database Support for Object-Oriented Programming. In *Advances in Object-Oriented Database Sytems, 2nd International Workshop on Object-Oriented Database Systems* (Bad Münster am Stein-Ebernburg, FRG, Sept. 1988), Springer-Verlag, pp. 23–42.

[15] *Maier, D., and Stein, J. Development and Implementation of an Object-Oriented DBMS.* Computer Science Series. The MIT Press, Cambridge, Ma, 1987, pp. 355–392.

[16] *Peterson, R. W., Bannon, T., Esralian, P., Gupta, A., Wang, C., and Wells, D.* Persistence for C++ Using Zeitgeist. Technical Report ITL-89-10-02, Rel. 0.1.3, Texas Instruments Incorporated, P.O. Box 655474, MS 238, Dallas, TX 75265, Jan. 1990.

[17] *Rosenthal, A., Heiler, S., Dayal, U., and Manola, F.* Traversal Recursion: A Practical Approach to Supporting Recursive Applications. In *Proceedings of ACM-SIGMOD '86 International Conference on Management of Data* (Washington, D.C., 1986), pp. 166–175.

[18] *Shaw, G. M., and Zdonik, S. B.* An object-oriented query algebra. In *Proceedings of the Second International Workshop on Database Programming Languages* (June 1989).

[19] *Shaw, G. M., and Zdonik, S. B.* A query algebra for object-oriented databases. In *Proceedings of the Sixth International Conference on Data Engineering* (Jan. 1990), pp. 154–162.

[20] *Stonebraker, M., and Rowe, L. A.* The Design of POSTGRES. In *Proceedings of ACM-SIGMOD '86 International Conference on Management of Data* (Washington, D.C., 1986), pp. 340–355.

[21] *Stonebraker, M., Wong, E., Kreps, P., and Held, G.* The Design and Implementation of INGRES. *ACM Transactions on Database Systems 1*, 3 (Sept. 1976), 189–222.

[22] *Stroustrup, B. The C++ Programming Language.* Addison-Wessley, 1986.

[23] *Stroustrup, B.* Parameterized Types for C++. In *Proceedings of the USENIX C++ Conference* (Denver, CO, Oct. 1988), pp. 1–18.

[24] *Zaniolo, C.* The database language GEM. In *Proceedings of ACM-SIGMOD 1983 International Conference on Management of Data* (San Jose, CA, May 1983), pp. 207–218.

Important Features of Iris OSQL

William Kent

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303

1 Introduction

OSQL is the object-oriented database language being developed for the Iris object-oriented database management system at Hewlett-Packard Laboratories [Fishman89].

This paper provides an overview of the underlying concepts and some important features of OSQL in the current prototype. The language is still under development. Not all features are presented here, and those presented are subject to change.

2 The Iris Data Model

The Iris Database System is based on a semantic data model that supports abstract data types. Its roots can be found in previous work on Daplex [Shipman81] and Taxis [Mylopoulos80]. A number of recent data models, such as PDM [Manola86] and Fugue [Heiler88], also share many similarities with the Iris Data Model. The Iris data model contains three important constructs: *objects*, *types* and *functions*. These are briefly described below. A more complete description of the Iris Data Model and the Iris DBMS may be found in [Lyngbaek86, Fishman89].

2.1 Objects and Types

Objects in Iris represent entities and concepts from the application domain being modeled. Some objects such as integers, strings, and lists are self identifying. Those are called *literal* objects. There is a fixed set of literal objects, that is each literal type has a fixed extension.

A *surrogate* object is represented by a system-generated, unique, immutable *object identifier* or *oid*. Examples of surrogate objects include system objects, such as types and functions, and user objects, such as employees and departments.

Types have unique names and are used to categorize objects into sets that are capable of participating in a specific set of functions. Objects serve as arguments to functions and may be returned as results of functions. A function may only be applied to objects that have the types required by the function.

Types are organized in an acyclic type graph that represents generalization and specialization. The type graph models inheritance in Iris. A type may be declared to be a subtype of other types (its supertypes). A function defined on a given type is also defined on all its subtypes. Objects that are instances of a type are also instances of its supertypes.

User objects may belong to any set of user defined types. In addition, objects may gain and lose types dynamically. For example, an object representing a given person may be created as an

[0] ©1990 Hewlett-Packard Company

instance of the Employee type. Later it may lose the Employee type and acquire the type Retiree. When that happens, all the functions defined on Retiree become applicable to the object and the functions on Employee become inapplicable.

2.2 Functions

Attributes of objects, relationships among objects, and computations on objects are expressed in terms of functions. Iris functions are defined over types, they may take multiple arguments, they may be many-valued and, unlike mathematical functions, they may have side-effects. In Iris, the declaration of a function is separated from its implementation, providing a degree of data independence.

Function names may be overloaded, i.e., functions defined on different types may be given identical names. When a function call is issued using an overloaded function name, a specific function is selected for invocation. Iris chooses the function that is defined on the most specific types of the actual arguments.

The functions applicable to an object are determined by the functions defined on all of its types. A type can be characterized by the collection of functions defined on it, or, more precisely, by its participation in functions of one or more arguments. Thus the Employee type might have the EmpNo and Salary functions defined on it. It might be further characterized as being the first argument in an AsgmtDate function, which returns the date a given employee was assigned to a given department.

2.3 Database Updates and Retrievals

Iris object semantics are described entirely in terms of the behavior of functions, i.e., the results returned by their application. Users are shielded from the data structures which internally implement these behaviors. As a simple example, function values might be stored in various configurations of relational tables without affecting their semantic behavior.

Updates, or *side effects*, are described in terms of altering the results returned by functions. Properties of objects and relationships among objects are modified by changing the values of functions, using `set`, `add`, and `remove` statements. A *procedure* is a function whose implementation has side effects.

The database can be queried by using the OSQL `select` statement, described later.

2.4 Function Implementation

Before a function may be invoked, an *implementation* must be specified. The implementation of the function is compiled and optimized into an internal format that is then stored in the system catalog. When the function is later invoked, the compiled representation is retrieved and interpreted.

Iris supports three methods of function implementation: Stored, Derived, and Foreign.

The extension of a stored function is maintained as stored data.

Derived functions are functions that are computed by evaluating an Iris expression. The expression may represent a retrieval or an update. A function may also be derived as a sequence of updates, which defines a procedure.

A foreign function is implemented as a subroutine written in some general-purpose programming language and compiled outside of Iris. The implementation of the foreign function must adhere to certain interface conventions. Beyond that, the implementation is a black box with respect to the rest of the system.

2.5 Iris System Objects

In Iris, types and functions are also objects. They are instances of the system types, `Type` and `Function`, respectively. Like user-defined types, system types have functions defined on them. The collection of system types and functions model the Iris metadata and the Iris data model operations.

System functions are used to retrieve and update metadata and user data. Examples of retrieval functions include, `FunctionArgcount`, that returns the number of arguments of a function, `SubTypes`, that returns the subtypes of a type, and `FunctionBody`, that retrieves the compiled representation of a function. *System procedures* correspond to the operations of the data model and are used to update metadata and user data. Examples of system procedures include `ObjectCreate`, to create a new object, `FunctionDelete`, to delete a function and `IndexCreate`, to create an index.

3 An OSQL Sampler

The statements

```
create type Person;
create Person instance :allen;
```

create the type Person and one instance, whose object identifier is returned in the *host variable* `:allen`. We can't do much more than query the type of this object, or the instances of the Person type, until we introduce some functions:

```
create function Name(Person) -> String;
create function SocSecNum(Person) -> Integer;
create function Hobbies(Person) ->> String;
```

These three functions each take a Person as argument, and return results of the indicated types. The `->>` signifies that the function is multi-valued, i.e., its result is a set of objects.

The following statements add and remove values of functions:

```
set Hobbies(:allen) = {'skiing','fishing'};
set Name(:allen) = 'Allen';
set SocSecNum(:allen) = 999999999;
add Hobbies(:allen) = 'chess';
remove Hobbies(:allen) = 'fishing';
```

These statements have illustrated some fundamental concepts, but OSQL provides more elegant alternatives. Let's first undo what we've done:

```
delete :allen;
delete Type Person;
```

The first statement deletes Allen; all function applications taking that object as argument or result are thereafter undefined. The second statement deletes the Person type itself. All functions created with that type as argument or result are also deleted.

Now we restart, getting equivalent results by creating a type using *in-line* definitions of *property functions* (functions of one argument):

```
create type Person
properties
(
 Name String,
 SocSecNum Integer,
 Hobbies ->> String
);
```

That's semantically equivalent to the four separate statements used earlier to create the type and three functions. Any number of property functions may be defined in-line, and any number may be created separately as before.

Object creation can be bundled, creating several objects at a time, and also initializing some property values:

```
create Person
properties (Name, SocSecNum, Hobbies)
instances
  :allen ('Allen', 999999999, {'skiing','chess'}),
  :brian ('Brian', 888888888, 'piano');
```

The host variables :allen and :brian now contain oid's of these newly created objects. The current value of SocSecNum(:brian) is 888888888.

Let's introduce some more types:

```
create type Stockholder
properties (NumShares Integer);

create type Employee subtype of Person, Stockholder
properties
(
 EmpNo Integer,
 Salary Integer
);
```

By these definitions, an employee is a person and also a stockholder. All the functions defined on persons and stockholders are also defined on (*inherited* by) employees. When a new employee object is created, any of those functions may be initialized:

```
create Employee
properties (EmpNo, Salary, Name, SocSecNum, NumShares)
instances
  :carla (0001, 50000, 'Carla', 777777777, 200);
```

Allen can become an employee by acquiring a new type:

```
add type Employee to :allen;
```

That also makes him a stockholder, and we can

```
set EmpNo(:allen) = 0002;
set Salary(:allen) = 35000;
set NumShares(:allen) = 100;
```

Let's make Brian an employee, too, taking another shortcut:

```
add type Employee
properties (EmpNo, Salary, NumShares)
to :brian (0003, 40000, 100);
```

We need departments:

```
create type Department
properties (DeptNo Integer);

create Department
properties (DeptNo)
instances :toy (901);

create function CurrentDept(Employee) -> Department;

set CurrentDept(:carla) = :toy;
```

Our first look at a query, and a function derivation:

```
create function CurrentEmps(Department d) -> Employee e
as
select each Employee e where CurrentDept(e)=d;
```

That defines CurrentEmps of a department as all employees who have that department as their CurrentDept.

In certain cases, when a derived function is easily recognized as a simple inverse of a stored function, the derived function can be updated. We can put everybody in the toy department by

```
add CurrentEmps(:toy) = {:allen, :brian};
```

We can fire Allen, without deleting him from the database, by removing the Employee type:

```
remove type Employee from :allen;
```

All the functions defined on Employee are no longer applicable to Allen; he no longer has an employee number or salary. All the functions having Employee as a result type no longer have Allen in their results; Allen will not be included in `CurrentEmps(:toy)`.

Bulk updates, resembling queries, can be done. If we had a Manager function defined on employees, the following would set the salary of each employee to the salary of his manager:

```
set Salary(e) = s
for each Employee e, Integer s
where s = Salary(Manager(e));
```

Notice that the semantics of this statement is to update the salary of each employee to the old salary (e.g., the salary that prevailed immediately before the update) of his or her manager.

Here's a function with multiple arguments:

```
create function
AsgmtDate(Employee e, Department d) -> String date;
```

This function "belongs" to both the Employee and Department types. It is not a property function, and can't be defined in-line.

```
set AsgmtDate(:carla,:toy) = '5/1/90';
```

Here's a function with complex results:

```
create function
AsgmtHist(Employee e) -> <Department d, String date>
as
select each Department d, String date
where AsgmtDate(e,d)=date;
```

Assuming more data than we've actually shown in these examples, the value of `AsgmtHist(:carla)` might be the set of lists

```
{
 <@sales, '2/4/87'>
 <@toy,   '5/1/90'>
}
```

in which we use `@sales` and `@toy` to denote the oid's of those two departments.

Function overloading can be illustrated in a simple way by imagining that employees have more stockholder votes than ordinary stockholders, getting an additional vote for each week of employment. We might then have the Votes function defined twice:

```
create function Votes(Stockholder s) -> Integer v
as Select NumShares(s);

create function Votes(Employee e) -> Integer v
as Select Sum(NumShares(e),WeeksEmployed(e));
```

While Allen was an employee, `Votes(:allen)` was computed by the second formula. After we fired him, it is computed by the first.

4 More About Queries

The Select and Cursor statements return Iris objects. Selects return all items that match the specified criteria at once. Cursors provide control over how to return the result of a query.

4.1 Select

The general form of a select statement is:

```
SELECT result-spec
FOR EACH source-spec
WHERE filter
```

The *source-spec* specifies a list of *select variables* and the domain of each. A domain may be

- a type, in which case the variable ranges over all instances of the type;
- `bag of` *type*, in which case the variable ranges over all bags of instances of the type.

The *filter* (also known as the *predicate-expression*) gives the criteria to be used in the search. The *result-spec* defines the objects or list of objects to be returned in the results; it may include functions to be applied.

The result of a Select is a *bag* containing atomic values or lists of atomic values. The bag has no persistent ordering. Duplicates may occur, unless suppressed by using the `distinct` keyword. Non-literal objects in the result are represented by their oid's.

If the query returns a single result (either atomic or a list), an **into** option is available for specifying interface variables into which the values of the result will be placed. It is also possible to assign the result of a select-expression to host variables for later use.

The semantics of evaluating a select-expression can be described in four steps:

1. Form the cross-product of the domains of the select variables. The cross-product is a set of lists, each containing a member of the domain of each variable.
2. Apply the specified predicate-expression to each potential result in the cross-product. This might result in the elimination or replication of lists in the cross-product.
3. Generate the result list(s) by applying the *result-spec* (e.g., functions) to the occurrence of each result satisfying the predicate-expression.
4. Eliminate duplicates if the keyword `distinct` is specified.

The *result-spec* may be empty, as in

```
select each Employee e
where Salary(e)>50000;
```

The semantics of this abbreviated form are that all lists from the cross-product which satisfy the predicate-expression are returned. In effect, the *source-spec* doubles as the *result-spec*.

In general, a predicate-expression is a conjunction and/or concatenation of comparisons. Variables, constants, function results and lists may be compared, and used as a *filter* to weed out unwanted values.

The form `y=f(x)` is overloaded. It is true in the following cases:

- y is a bag variable and f is multi-valued, such that the bag y contains all the objects in the result of f(x).
- y is a non-bag variable and f is single-valued, such that the value of y is the same as f(x).
- y is a non-bag variable and f is multi-valued, such that the value of y is a member of the result of f(x).

Functions may be applied to variables and constants and to the result of other functions.

The logical operators currently supported are **and** and `concat`. The `concat` operator is equivalent to a union without removal of duplicates.

The following queries are equivalent, each returning the names of all employees and their managers:

```
select ne, nm
for each Employee e, Employee m, String ne, String nm
where m=Manager(e) and ne=Name(e) and nm=Name(m);
```

```
select ne, nm
for each Employee e, String ne, String nm
where ne=Name(e) and nm=Name(Manager(e));

select Name(e), Name(Manager(e))
for each Employee e;
```

The following query returns the names of all people whose hobbies include both skiing and chess:

```
select Name(p)
for each Person p
where Hobbies(p)='skiing' and Hobbies(p)='chess';
```

No person will be represented in the result more than once, though there may be duplicates in the result if several people had the same name.

The following query returns the names of all people whose hobbies include either skiing or chess:

```
select Name(p)
for each Person p
where Hobbies(p)='skiing' concat Hobbies(p)='chess';
```

In this case, a person whose hobbies include both skiing and chess will occur twice.

Using bag variables and the system function Count, we can count the number of hobbies each person has:

```
select Name(p), n
for each Person p, Integer n, bag of String h
where h=Hobbies(p) and n=Count(h);
```

The bag variable is implicit in the following equivalent query:

```
select Name(p), Count(Hobbies(p))
for each Person p;
```

Cursors allow the results of a Select statement to be obtained one at a time and assigned to host variables for later use, or several at a time for display.

To open a cursor called :esal for the retrieval of employee names and salaries:

```
open :esal for
select Name(e), Salary(e)
for each Employee e;
```

To fetch the names and salaries of employees one at a time into variables:

```
fetch :esal into :emp1, :sal1;
fetch :esal into :emp2, :sal2;
...
```

To display the names and salaries of the next ten employees:

```
fetch :esal next 10;
```

When we're finished:

```
close :esal;
```

5 Conclusion

This paper has described some of the main features of Iris OSQL, illustrating its overall semantic approach to object-oriented database.

References

[Fishman89] D. H. Fishman et al. Overview of the Iris DBMS. In W. Kim, F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications.* ACM Press, New York, N.Y., 1989.

[Heiler88] S. Heiler and S. Zdonik. Views, Data Abstraction, and Inheritance in the FUGUE Data Model. In Klaus Dittrich, editor, *Lecture Notes in Computer Science 334, Advances in Object-Oriented Database Systems.* Springer-Verlag, September 1988.

[Lyngbaek86] P. Lyngbaek and W. Kent. A Data Modeling Methodology for the Design and Implementation of Information Systems. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

[Mylopoulos80] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database Systems*, 5(2), June 1980.

[Manola86] F. Manola and U. Dayal. PDM: An Object-Oriented Data Model. In *Proceedings of 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

[Shipman81] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), September 1981.

Strawman Reference Model for Change Management of Objects

John Joseph
Mark Shadowens
John Chen
Craig Thompson

Texas Instruments Incorporated *†‡

Abstract

This paper provides a strawman[1] reference model which can be used for comparing and reasoning about change management systems. It begins with a glossary of change management terms, providing common ground for *people* to communicate about change management. A *descriptive* reference model is then described as consisting of a collection of characteristics that can be used, again by people, for comparing existing and future change management systems. Finally, based on the descriptive reference model, a *functional reference model* is defined that provides a much more precise description of how *machines* can realize generic change management operations. While the glossary and descriptive reference model are steps on the path to forming standards, the more precise model is closer to what is needed to provide interoperability (standards) for machines. It is hoped that the step-wise refinement exposition used in this paper, from terms through a descriptive reference model to an operational reference model, will provide a good roadmap for OODBTG or similar groups to reach consensus leading to standards in the area of change management.

*Information Technologies Laboratory, Computer Science Center, Texas Instruments Incorporated, P.O Box 655474, MS 238, Dallas, Texas 75265.

†Email: joseph@csc.ti.com Telephone: (214) 995-0305

‡Comments are welcome!

[1] This paper is intended as a good start towards a change management reference model; its treatment of change management is not intended to be complete. It is written in such a way that it is consistent with OODBTG's OODB Reference Model. It could be viewed as a mini-reference model refining the Change Management section of that reference model. Alternatively, it can stand alone as a separate reference model since it identifies and isolates an orthogonal abstraction.

1 Introduction

1.1 Motivation

Many application areas cannot be modeled in an adequate manner by representing only the present state of the world. Generally, support for modeling this aspect of an application is left to the application designer. But, it is now recognized that lifecycle support requirements for managing change in areas like computer-aided design (CAD), software engineering (including CASE), and cooperative authoring are quite similar. Thus, we can consider providing common support at the system level, providing generic ways to represent and manipulate complex objects, their interrelationships, and their histories.

The above applications need support for the evolution of objects and for constructing and configuring objects into layered, hierarchical descriptions. Information is represented in multiple forms at different stages in the evolution of applications. These multiple forms correspond to different levels of abstraction and transformations among these abstractions are often necessary. Applications need support for managing consistency across such transformations. We define *change management* as a consistent set of techniques that aid in evolving the design and implementation of an abstraction. These techniques can be applied, by cooperating users, at different levels to record history, reconstruct past state, explore alternatives, navigate through layered hierarchical designs, and manage multiple sets of representations of the design information.

Application information includes not only the data used in the applications but also the schema which structures or models the information. In a database, the schema controls the creation and manipulation of persistent objects. The transient application schema and data and the persistent

database schema and data are often the same in object oriented applications. Thus, change management is going to be needed by object-oriented applications and databases to manage change uniformly.

1.2 Outline of the Paper

Section 2 defines a glossary of (many of the) terms that people use in the area of change management to describe and communicate about the basic concepts. Section 3 is the descriptive reference model for change management systems. It identifies characteristics or properties that any change management system must address. Section 4 surveys several existing change management systems and compares them with respect to the features identified in section 3. Finally, in section 5, we define a precise functional model for change management. The model can be realized as a collection of three *abstract machines*. We have prototype implementations of two of these machines. These prototypes are described in the appendices.

2 Glossary of Terms

The glossary defines terms used in the change management area. It is provided to ensure we are all talking the same language. It also helps to scope the concept terrain for change management. Below, we use a conceptual grouping, placing related terms near each other.

abstract machine A technique for defining systems and designing software. In this technique, a software system is composed of *state*, or data, and a set of *operations* or instructions which manipulate the machine state.

change management A consistent set of techniques that aid in evolution, composition and policy management of the design and implementation of an object or system.

version management A consistent set of techniques which aid in the management of the evolution of an object.

version control See *version management.*

configuration management A consistent set of techniques which aid in the management of the composition of an object.

configuration control See *configuration management*

transformation management A consistent set of techniques which aid in the management of the dependencies and constraints which exist among objects.

transformation control See *transformation management.*

constraint management See *transformation management.*

dependency tracking See *transformation management.*

administrator The person or persons invoking change management over a set of objects or system.

version A variant of the original value of an object.

version graph The data model used to represent the evolution of versions of an object.

version node A node on a version graph representing a version of an object.

child version A version of a version of an object.

immediate version See *child version.*

children versions A set of child versions.

parent version A version from which a certain child version (or children) evolved.

parents versions A set of parent versions.

primary version The member of the children versions designated by the administrator to be the chief representative of the children versions.

major version See *primary version.*

default version See *primary version.*

alternate version The members of the children versions which are not the primary version.

minor version See *alternate version.*

sibling versions The versions which are members of the same set of children versions.

initial version See *root version*

root version The initial value of an object.

original version See *root version.*

working version A local copy of a released version of an object that is currently undergoing modification.

experimental version See *working version.*

checkout The transaction which creates a working version.

effective version A version of an object that is not modifiable and represents an object suitable for testing prior to its actual release.

checkpoint See *effective version.*

checkin The transaction which creates an effective version.

released version A version of an object that is not modifiable and has passed all evaluation tests and has been archived.

baseline See *released version.*

next version The primary version of a version's children

previous version The parent version or parents of a version.

linear version The policy that a version can only have a single child version associated with it.

branching versions The policy that a version can have more than one child version associated with it.

version history See *version graph.*

merging versions Sibling versions who share a common child version.

timestamp A method for marking or uniquely identifying a version chronologically.

delta (version) The information which differentiates a version from members of its immediate family.

forward delta The delta which , when combined with a version, creates a child version.

backward delta The delta which , when combined with a version, creates a parent version.

configuration graph The data model used to represent the composition of an object.

configuration node A node on a configuration graph representing an object component.

component The data model abstraction of an object.

supercomponent A component which can be decomposed into a set of subcomponents.

subcomponent A component which can be composed with other components to form a supercomponent.

abstract configuration The configuration graph of an object in which only the structure of the composition has been defined without the association of configuration nodes with component objects.

dynamic configuration The configuration graph of an object in which the version of some components or configuration nodes have not been bound.

concrete configurations The configuration graph of an object in which each version of a component or configuration node has been bound.

multiple representations More than one aspect or combination of aspects representing an object or view of an object.

transformation The manner of mapping one representation of an object to another representation of the object.

dependency The manner of determining the relationship of objects.

constraint A rule or manner of being or interaction of an object that is stated and enforced.

policy A statement of constraint or dependency.

aggregation The composition of objects.

granularity The level or degree of aggregation of a set of objects.

3 Descriptive Reference Model for Change Management

This section presents a descriptive reference model for change management. The intent is to isolate the key concepts that make up the change management functional abstraction in computer systems. The

Granularity	Flexible, Application level objects
Polymorphism	With respect to object types and storage status
Versions	Branching, merging, baselining
Deltas	Forward, backward, delta-sets
Configurations	Abstract, concrete, contexts
Multiple Representations	Consistency maintenance, transformations
Transformations	Constraint Enforcement, Partial consistency
Dependancies	Dependancy analysis, validation, verfication
Pay as You Need	Modular system, no penalty for unused components
Documentation	Design decisions, change reasons, dependencies
Policies	Policy layer, organizational and change management policies
Security and Authorization	Lock, own , grant and share access
Multiple Users	Co-operative work, nested transactions, concurrency control
Distributed Design	Multi-server, multi-client, replication consistency
Persistence	Deal with persistent and transient data in a uniform manner
Usability	Non-intrusive, low-profile, user interface

Table 1: Change Management Requirements

approach taken is to determine a minimal basis set of features or characteristics that change management systems must address [2]. Viewed from another perspective, these characteristics can also be viewed as requirements for change management that have been identified from a variety of applications.

In this section, we discuss the factors involved in change management. We begin with a discussion of the characteristics of design objects such as granularity, interface and implementation, and different views. Changes on objects which cause different versions and configurations are then discussed. We then address the issues related to the validation of designs; namely, transformations and dependencies. At the end we cover the issues more relevant to end users such as query facilities and performance. These requirements may be classified as the requirements of the logical model(3.1 - 3.11) and the requirements of the operational model(3.12 - 3.16). It is expected that the application environment provides tools to support the operational model. These latter requirements are typically independent of change management and can be separately specified.

Table 1 provides a summary of the characteristics identified. The change mangement systems referred to in this section are described in Section 4.

[2] Often, these characteristics are addressed implicitly, as when a change management system only provides builtin data types like module or file.

3.1 Granularity of Objects

In today's software systems, change management systems often interact closely with file systems and only support files as design objects. The information needed for change management is kept partially by the file system, and partially by either a conventional DBMS or by the change management system itself. The file system also keeps the objects persistent on the secondary storage. Tools available in the supporting environment, such as editors, can often be used with the change management system since they all operate on files. However, often files are not the right level for change management. A file is usually composed of a number of logical units such as design blocks, function definitions, or sections of text. From another angle, a file is just an artifact of the storage subsystem. A change mangement system should not be tied into the storage system; it should instead deal with the logical units, namely objects, of the application. It should be able to manage change at the level of granularity that the application requires. It is the responsibility of the application to select the grain size of objects to be change-managed. It should be possible to manage change at multiple levels of granularity, including not only containing objects but contained objects, recursively. This selection will impact the performance and flexibility that the application provides to its users. The change management system should not limit the application in this selection.

3.2 Polymorphism

A general-purpose change mangement system must be able to deal with user-defined types of objects. In many systems, the type of objects managed is restricted to a few built-in types like lines of text or Lisp S-expressions.

3.3 Versions

A design object is represented by many *versions* during its lifecycle. Often the interface of the object is invariant and the implementation changes over time.

The version derivation sequence of an object reflects how the object evolves. The simplest way an object could evolve is linear: changes to the object always occur on the current version. Support for linear changes is provided by most data management systems. In many applications, however, changes often occur in a non-linear fashion. A designer may want to explore several design alternatives first, and then select one. A change management system needs to support alternatives or *branching versions*. The desired design object may be selected from a combination of several possibilities, implying that the system needs to support *merging* of versions. Version merging is also important for supporting a group of designers performing parallel development. *Version baselining* is often desired, that is, recording a system configuration at specific versions, evolving the configuration through several changes, but at any point being able to return to any recorded version. This is convenient for supporting released or demo designs that must be available in their original state, but can also be evolved.

It is not necessary that the evolution of objects is along a temporal dimension. Versions can be created based on other qualitative or quantitative attributes meaningful to the application. Since design objects are usually structured hierarchically, the creation of versions of object can trigger versions of other objects in the hierarchy to be generated. There are cases when the designer doesn't want a small change to a low-level object to ripple through an entire hierarchy creating a new version of everything above it. Hence it is desirable to allow the designer to control the triggered creation of versions on objects[32].

3.4 Changes and Change Groups

When an object evolves, new versions of the object are created. The *changes* made to the object (sometimes referred to as *deltas* or *differential representations*) are an important part of the design history. If the object is big and/or the evolution is frequent, the space needed to keep all the object versions becomes intolerably large. Since the difference between object versions may be relatively small compared to the size of object versions, people have used changes to represent object versions.

Changes can be forward-going or backward-going. Forward-going changes make it expensive to obtain newer versions, but they make it easy to support version branching. On the other hand, backward-going changes make it expensive to obtain older versions. Since designers are most likely to work on more recent versions, backward-going changes are often preferred. A change management system can provide support for both. Also, in both schemes, periodic full versions may be preserved to trade off storage space for computations.

Many change management systems provide a mechanism which allows the designer to tie related changes into change groups (or delta-sets). The designer, thus, can treat related changes as a unit and back out of all these changes at once when they don't work. PIE allows the user to include/exclude change groups when making a new design. A change group is also an appropriate place to keep associated documentation. High-level documentation can then be derived and used in the development history which is useful in maintenance. CLF further allows the designer to classify the change groups and query the change groups based on their classifications.

3.5 Configurations

Objects can be primitive or composite. A composite object is composed of other objects. A specification of the composition of an object is a configuration. The relationship between an object and its components may be bound on a range of levels:

Versions The versions of components are specified, as needed in a released design.

Interfaces The interfaces of objects are specified. Any object version which satisfies the interface requirements can be used.

Contexts A context is a sequence of change groups. This allows the designer to specify object instances in terms of change groups.

Representations and/or host machines Only the representations and/or host machines are specified. This is useful in specifying generic or representation-independent objects, and distributed objects.

A configuration in which the components are not bound, but specified according to a characterestic is called an abstract or dynamic configuration. When the components are bound to specific objects, the configuration is referred to as concrete. A change management system is required to support dynamic configurations of objects with respect to user specified characterestics of an application as well as concrete configurations.

3.6 Multiple Representations

The lifecycle of a design involves different representations of data at different design stages. These representations may correspond to different levels of abstraction or different viewpoints. In VLSI CAD, a circuit can be represented in register level description, gate level schematics, and layout geometry. Since these representations all correspond to the same design object, consistency must be maintained among them. When changes are initiated in different representations, consistency management becomes more difficult, since changes in representations which are at lower levels of abstraction are often not transmitted back to higher level representations, causing inconsistencies.

Multiple representations may be managed in two different ways. One way is to designate a primary representation for changes. Other representations are automatically generated from it. Most change management systems for software development address the consistency problem in this way. They designate the source code as the primary representation for changes. This approach imposes a severe restriction on the relationship among design representations: all representations must be automatically generated from a primary representation. Since in general, it is not possible for a high-level representation to be automatically generated from a low-level representation, the changes can only be made at the highest-level representation.

The second way of maintaining the consistency among different representations is to provide the designer with interactive tools so that changes on a lower-level representation can be tracked and actions to be applied to higher-level representations can be recorded and presented to the designer. This approach[4] to maintaining consistency is more general as it allows changes made in one representation to be propagated to other representations, automating automatable transformations and at least bookkeeping where manual transformations are needed.

3.7 Transformations

Operations applied to objects during their lifecycle are transformations. There are transformations on single object representation to achieve specific goals, such as editing, simulation, and analysis. There are transformations to bring objects from a higher representation to a lower representation, such as translation. Since a design is usually developed jointly by different designers on different machines, and design tools are available on different machines, there are transformations to transport objects across machines. As a result of transformations, new objects may be created. These new objects are different from the original objects in versions, representations, or host machines.

A major function of change management systems is to enforce certain pre-defined constraints among transformations and objects. The constraints may specify when a transformation can take place or how the transformations are ordered. For example, a compilation transformation is performed when the source code has a newer timestamp than the binary code; or a transformation of loading binary code always follows the transformation of compiling the corresponding source code, when both transformations need to be performed. There are cases where it is desirable to have transformations generated by the change management system. This allows the user to work with higher-level development operations.

3.8 Dependencies

Objects are not independent of one another. The dependencies among objects often reflect on transformations. For example, when a macro of a software system is modified and re-compiled, the functions which use the macro should also be recompiled. Some change management systems allow the designer to specify the dependencies. They then generate the implied transformations when needed. Common-Lisp Framework (CLF) has a static analysis tool which can analyze the dependencies of objects. The Source Code Control System developed by James Rice (Rice's SCCS) provides a way of enforcing transitive dependencies.

One of the most difficult tasks for a designer modifying a complex design is for him to understand the effects of his change. Validation can be thought of as satisfaction of a constraint. A change management system is required to support full and partial validation of application systems.

3.9 Pay As You Need

An application may not need all services of a change management system. The system should be configurable in such a way that an application is not forced to bear the overhead of a service that it does not need. A partitioning of the change management functions into logical groups which can be put together in a "LEGO" fashion will support such a requirement. The change mangement model proposed in Section 5 follows this approach.

3.10 Documentation

Design decisions, reasons for changes, configurations, or dependencies need to be recorded. This information can serve as the basis for communication of design rationale among the designers. It also provides the traceability on design evolution, which is useful and is often needed for legal accountability.

3.11 Policies

The policies of an organization regarding change management evolve for legal, accounting or other reasons. Common cases are policies regarding audit trails, releases, and validation items. It is necessary for a change management system to support these policy changes without a major perturbance to applications. This is possible if the policies are managed by a flexible policy layer. It may be necessary to have policy sub-layers based on applications, sites or enforcement types. This policy layer can also be used to manage the evolution of policies internal to the change management system; examples are policies regarding lazy/eager evaluation, caching/memoization, clustering hints to storage managers, and use of deltas.

3.12 Security and Authorization

It must be possible to lock, own, grant, share, and limit access to objects that are being change-managed. In most systems, these capabilities are inherited from the database or file system that the change management system interacts with. In some systems, it is possible for users to bypass the change management system and access change-managed objects directly. This is dangerous since version histories and other change management state may not be preserved, unless some triggering scheme detects the change and alerts the change management system.

3.13 Multiple Users

Designs are often created by a group of designers working as a team. In certain cases, different designers work on disjoint parts of a design; but frequently, designers need to interact with each other and work on common design objects. Therefore, change management systems need to support the concurrent development of shared design objects in a coordinated manner. Each unit of development, called a *design transaction*, brings a design from a consistent version to a new consistent version[3]. Design transactions are long duration and ended by *releases* of various levels: to the designer himself, for testing with other changes, for validation, or for distribution. To ensure the integrity of shared design objects across transactions, some mechanism for concurrency control is needed. For the reasons of security and better management of shared objects, issues such as the protection on design objects and the authorization of operations on them need to be resolved.

The most primitive concurrency control mechanism is manual control. Designers avoid conflicts by conversational arrangements. This might be acceptable for a small design group physically located in the same area when the design objects are small enough so that concurrent modifications on the same objects rarely happen. In general, this kind of concurrency control is not appropriate. Change management systems need to support cooperative work and manage parallel development. It may be necessary to devise protection and authorization schemes to guard against accidental or intentional misuse of common data.

[3] The notion of globally consistent state used in conventional databases is inappropriate since in designs, the detailed design does not match the requirements until the design is completed. Instead, in design transactions, consistency is with respect to a suite of validation tests that measure whether sub-designs and design representations are interconsistent.

3.14 Distributed Design

In many applications, different components of the application tool set are often available on different kinds of machines. Therefore, objects need to be transferred across machines. Various design objects may also be created and developed on different machines. To improve availability, replicated copies are used on different machines. These copies need to be maintained consistent. In big projects, application data can be distributed in different geographic locations. Change management systems need to provide support for distributed objects, including uniform access and information recording, so that these objects can be used transparently by the designer on any machine.

3.15 Persistence

A change mangement system should be able to deal with both transient and persistent objects in a uniform manner. Traditionally, these systems have supported persistent objects only. Transient objects also need support for evolution, configuration into structured objects, and transformations. "What if" experimentation and constraint management for transient objects should be supported. Persistence is expected to be provided separately in the environment; change mangement systems are required to interface to persistent objects as well as be able to use the provided persistence for storing the change management information.

3.16 Usability

Usability of a change management system is an important factor since such systems will only be adopted if they do not get in the way of progress. One major criterion in determining the usability of a system is *execution efficiency*.

Another criterion is *integration*. Change management users perceive that there is large overhead if the change management system is intrusive or high profile. The ideal situation is when the user interface of an application hides the change mangement interface from the user by casting it in application terms.

The *user interface* of a change management system also affects the usability of the system. The user needs to have an easy way of specifying, viewing, and changing the definition of objects, managing the configurations, specifying and applying transformations, and enforcing constraints and tracking dependencies among component objects.

4 Review and Comparison of Existing Change Management Systems

This section is included to provide a kind of litmus test of the descriptive reference model presented in section 3. Using the characteristics identified in that model, it should be possible to compare existing change management systems. Section 4.1 reviews selected changc management systems. Table 2 lists a comparison of systems based on the descriptive reference model of Section 3. The comparison is based on published literature and is necessarily incomplete. Section 4.2 touches upon recent work related to change management.

4.1 Existing CM Systems

4.1.1 Unix Make

Unix Make[18] is a change management system which provides a primitive mechanism for enforcing *dependency* constraints among files. The constraints are based on the timestamps of the files specified by the user. Each constraint is composed of a target, a list of dependent files, and a sequence of actions. Unix Make ensures that all of the files on which the target depends exist and are up to date by executing appropriate actions. The constraints define a graph of dependencies. Unix Make does a depth-first search of this graph to determine what work is necessary. It adopts a simple mechanism to determine the sequence of actions and hence is easy for the user to understand and to use. However, it doesn't have support for versions and its timestamp-based constraints are restrictive.

4.1.2 Source Code Control System (SCCS)

SCCS[38] was developed by Bell Laboratories and runs on IBM OS/MVT and Unix. SCCS manages changes on text files. Versions of a text file are represented by forward deltas consisting of lines of text and are stored together with the modified lines. Versions are named according to creation order. The user can restore any version by specifying the version name or the time when the version was current. Limited version branching and merging are supported. Branches can be created from, and later

on merged back to, the default version. When merging takes place, change conflicts are signaled. SCCS can only manage single text files. The granularity of change is individual lines only, not logical chunks of text. SCCS is not easy to use on branching versions since version names are assigned rather randomly and the internal structure maintained by SCCS gets complex fast.

4.1.3 Revision Control System (RCS)

RCS[43] is a change management tool running on Unix. Similar to SCCS, RCS manages changes on text files. RCS differs from SCCS by supporting changes to a group of files; it supports both backward and forward deltas; it keeps changes separate from the managed text; and it has better performance in most cases. The user interface of RCS is simple and hence easy to use. However, its locking mechanism seems to be primitive.

4.1.4 MAKE-SYSTEM and PATCH Facility

MAKE-SYSTEM and PATCH[35] Facility are the change management tools on Lisp machines for software, especially Lisp program, development. The user specifies objects, transformations, and dependencies involved in a software system. Objects can be sub-systems, modules, and files. The user is allowed to specify transformations in partial orders and the conditions under which the transformations are executed. Based on user's specifications, MAKE-SYSTEM determines at run-time what transformations to execute and the sequence of execution. The PATCH Facility supports changes to the Lisp programs at the S-Expression level. It provides a primitive function for the user to document the changes. MAKE-SYSTEM only supports linear evolution of objects and assumes that all objects can be decomposed into files. It does not fully support hierarchically structured objects and has no mechanism to support multiple users. The Explorer CM-Patch Facility[6] is a tool on top of MAKE-SYSTEM and PATCH Facility and provides multiple-user/workstation support. Under CM-Patch Facility, a workstation is designated as the server machine and keeps the change management information. Systems under development and their component files are registered to the server. Users then send requests to the server to check files out when they need to work on them.

4.1.5 Source Code Control System (Rice's SCCS)

Source Code Control System[37] is a change management tool developed at Stanford University by James Rice for Lisp program development. It is built on top of MAKE-SYSTEM. Rice's SCCS provides a menu-based user interface so that the user is freed from learning the syntax of MAKE-SYSTEM. It has better support for hierarchically structured systems. Subsystems are allowed to share modules. It has a locking mechanism to avoid concurrent modifications to files. The locking mechanism is implemented at a low level and is easy to enforce. Three levels of privileges are provided so that operations can be granted to specific users. Dependency enforcement is emphasized, including inter/intra-subsystem dependencies and transitive/intransitive dependencies. Although versioning on systems is different from that supported by MAKE-SYSTEM, Rice's SCCS can only handle linear evolution. Among other uses, Rice's SCCS was used to keep Explorer and Symbolics releases of systems consistent.

4.1.6 Common-Lisp Framework (CLF)

CLF[13], developed at Information Sciences Institute, University of Southern California, is an object-oriented programming environment for the design, implementation, and maintenance of software written in Common-Lisp. CLF has four layers of abstraction:

Physical object management layer Objects reside in a uniformly accessible object-base in the virtual address space of a workstation. The objects can be accessed associatively.

Logical object management layer Objects are classified by type which determines their properties. Types are structured in a directed acyclic graph such that each successor type inherits the properties of its predecessor type. Each type is associated with a set of consistency conditions which regulate the properties of the objects of this type and procedures can be attached to objects and triggered at specified conditions.

Program development assistant layer Types of Lisp program development are defined. Lisp S-expressions are the primitive objects in this layer. They are grouped into modules which

are then grouped into systems. A system is associated with development steps which capture the linear evolution of programs.

User interface layer The user is able to create, retrieve, modify, compile, and load the primitive objects supported by the above layer. Facilities are provided to help the user construct queries on the software objects; an object browser is provided.

Though built on top of a general-purpose object management layer, CLF has several weaknesses. Only built-in types (Lisp S-expressions, modules, and systems) are managed, not generic objects. Systems cannot be hierarchically structured. Shared modules and dependencies across systems are not supported. Only linear system evolution is supported.

4.1.7 PIE

PIE[22,23] is an object-oriented environment implemented in Smalltalk that uses a description language to support the interactive development of programs. The PIE environment is based on a network of nodes which describe different types of entities such as methods, classes, categories of classes, specifications, and configurations of systems. Attributes of nodes are grouped into perspectives. Each perspective reflects a different view of the entity represented by the node. The values of attributes of a perspective are relative to a context. Alternative contexts are created to store different values for the attributes in a number of nodes. These contexts allow the user to examine or compare alternative designs without leaving the design environment. Since there is an explicit model of the differences between contexts, PIE can highlight differences and provide tools for merging alternative designs. To support the incremental development of a single alternative, a context is structured as a series of layers. The assignment of value to a property is done in a particular layer. Retrieval from a context is done by looking up the value of an attribute layer by layer. New layers may be created by the system or by users wanting to group related changes in the same layer. PIE uses contracts between nodes to describe the dependencies between different elements of a system.

Using contexts and layers, PIE supports linear and non-linear versions, grouping of related changes, and limited merging. Contracts provide a number of different mechanisms for describing and enforcing dependencies between system elements. The layered network database facilitates cooperative design by a group and coordinated, structured documentation.

4.1.8 Module Update Manager (MUM)

MUM[4] is a system, proposed in John Beetem's doctoral dissertation, which provides a general mechanism for maintaining consistency of multiple representations (of VLSI data) throughout the design process. In MUM, data management involves two aspects: (1) hierarchy management (2) managing isomorphic multiple representations. The basic idea in hierarchy management is for each design module to know the other modules that depend on it (the list of its parents) so that when the interface of the module changes, the parents can be notified. Isomorphic multiple representations is a very powerful concept. Beetem's approach is that starting with isomorphic representations, one can guarantee isomorphism through a system using action buffers and prompts buffers. When a representation is modified, the change operations are logged in an action buffer. These actions are then translated into prompts for other representations and these prompts are queued up for manual or automatic resolution by the system or designer respectively.

4.1.9 Molecular Object Model

Molecular Object Model[3] is proposed by Batory and Kim as an object model for VLSI designs. A molecular object has an interface description and an implementation description, where the former remains constant and the latter changes over time and thus has different versions. When an object is instantiated, only the interface is copied; no implementation of the object is specified. This allows a composite object to be parameterized, since any object with the same interface as specified in the component object can be used.

Molecular Object Model is notable in that it operationally defines "version" in terms of object interface and implementation.

4.1.10 Version Server

Version Server[28,5,29] was developed at the University of California, Berkeley, to support an object model for representing the evolution of a design database over time. It interfaces with existing design tools and can manage conventional files and

	Unix make	SCCS	RCS	MAKE-SYSTEM and PATCH	Rice 's SCCS	CLF	PIE	MUM	Molecular Obj. Model	Version Server	NSE
Object Types	files	text files	groups of text files	object module, file, system	object module, file, system	object and module	typed object	object	object	file, OCT object	file
Granularity	file	line	line	file, S-exp	file, S-exp	object			object	object	component
Versions	no	yes	branching	linear	linear	linear	branching		linear	yes	yes
Deltas	no	yes	yes	yes		yes				yes	
Configurations	yes		no	yes	yes	yes		yes	yes	yes	yes
Multiple Representations	no			no			yes	yes		yes	
Transformations	yes			yes		yes	yes	yes		yes	
Dependencies	yes			yes	yes		yes	yes		yes	yes
Documentation	no	yes		yes		yes					
Policies	yes			yes	yes	yes	yes				yes
Security	yes		yes	no	yes						yes
Multiple Users	no			no		no	yes				yes
Distributed Design	no			no		no					yes
Persistence	yes		yes	no		yes				yes	yes
Usability	no	no		no						yes	yes

Table 2: Comparisons of Change Management Systems

Oct[4] objects. The Version Server manages change of design objects in three orthogonal dimensions: versions, configurations, and equivalence relationships among multiple representations. The version history of an object forms a tree where each node is a version. The derivation sequence of versions is represented explicitly. A currency indicator is kept in each version history. Configurations can be static or dynamic. Dynamic configurations are supported by a layer and context mechanism, similar to that used in the Pie system. The Version Server has a PROLOG-based validation subsystem to enforce integrity relationships among multiple representations. The basic dimensions of Version Server can be applied recursively yielding composite versions, composite equivalences, and versions of equivalences. A Graph Browser[21] is used to display a interrelated views of versions, configurations and transformations.

4.1.11 Sun NSE

Network Software Environment (NSE)[15] is a software development platform to handle administration and co-ordination activities of a group during the entire software development life cycle. Software developers work in individual workspaces called *environments* concurrently. The environments are hierarchical. Merging of these development paths are done automatically, if possible, or with manual assistance. There is a notification facility so that object changes can be monitored network-wide by defining notification tasks. The notification tasks are performed when changes which trigger the tasks occur in the object anywhere in the network.

NSE takes an object-oriented approach to change management. There are three pre-defined object types called *files*, *targets*, and *components*. Additional user defined types can be specified. There is a policy layer to customize the behaviour of operations like merging.

4.2 Evolution of Ideas

The problem of change management has been independently studied in the context of software development[14] and engineering design [27,11]. Existing systems can be classified into three general approaches. The *software workbench* approach is characterized by loose collections of tools (for example, UNIX MAKE [18]) [35,38,43]. These tools are usually implemented on top of a file system. The *software assistant* approach is characterized by built-in change management integrated into the environment, as in the Common Lisp Framework (CLF) system [13] and the Ontos system [1]. The *software associate* approach is typified by laboratory systems like CHI [41] and KBEMACS [45] that try to provide knowledge-based tools for planning, semi-automatic programming, theorem proving, or verification. A history of the evolution of change management ideas over the last several years is given in [29].

A paper on VLSI CAD modeling [3] introduced the concept of *parameterized versions*. A parameterized version is a way to support dynamic binding of specific versions of component objects. In this model, a component object points to its version graph and the binding of the component object to a particular version can be left unspecified. A companion paper [11] introduced a context mechanism for supporting dynamic configuration binding. Contexts are also discussed in [23]. Rumbaugh [39] makes a significant contribution in the area of controlling change propagation across operations. His idea is to associate propagation attributes for operations with the relationships. An operation propagates from an object to its related object in a way specified by the propagation attribute. Change notification/propagation schemes are discussed in [12,11,24]. In [32], Landis introduced the idea of limiting of scope change through propagation of *significant changes*.

Consistency maintenance is the principal way that change management supports *reuse* of objects. A generic approach to manage multiple representations is outlined in [4]; also, see [10] for a system managing multiple representations for document generation in TEX.

Schema evolution is versioning of types; it can be seen as an application of the change management system where type definitions are the objects to be *versioned*. Inheritance imposes a *configuration* structure on the types. The user is concerned with the effect that a versioning of a type has on existing instances, regarding behaviour and structure, of the type; this is the domain of the *transformation* aspect of change management. Conceptually, therefore, a change management system covers all bases; but there are interesting practical issues because the effect of changing a type extend to objects of that type and programs that use objects of that type. The problem of type evolution in object-oriented

[4] Oct is the Data Manager for the Berkeley Synthesis System.

database environments is discussed in [40]. Schema evolution of transient objects is supported in CLOS. Orion[2] has a taxonomy of schema changes that occur in common design application; Orion supports schema changes which occur as a result of change to the definition of a type as well as change to the type lattice of the application.

We have listed transformations, dependencies and consistency of multiple representations as requirements for a change management system. What is required, in general, is a constraint specification and management component. A body of work exists in the area of constraint specification and enforcement [42,33,7,8,9]. Database research[16] is relevant to both the logical and operational aspects of change management systems. Integrity constraints, rule-based systems, transactions, and concurrency control are of interest; also, of particular interest is research on temporal databases.

During the 1988 OODB Workshop at OOPSLA'88, a panel with commercial and research participation addressed the status of change management in OODBs[26]. At that time (and even now), many OODB systems do not have fully implemented change mangement systems. Some have implemented limited schema evolution (change mangement of classes or types). There was some agreement that, architecturally, change management is a distinct functional layer in an OODB (Iris[46,19] may be an exception). At Sigmod'89, a panel on version management[30] concluded that consensus on change management terms and ideas may well be possible.

5 Functional Reference Model for Change Management

The Descriptive Reference Model of section 3 is useful for identifying the basic dimensions of a change management system, but it is imprecise. This section presents a more precise Functional Reference Model. While logical or other approaches could have been taken, we provide an abstract machine approach. An abstract machine is a specification of state and operations on that state that together defines how a system operates. The specification can then be implemented in one or more programming languages or in different systems to provide a uniform semantics for change management.

Traditionally, change management has been provided by domain specific software tightly bound to and bundled with the software application. This approach amounts to re-implementing the same set of techniques in each application. Oftentimes, change management support is not provided, or is inadequate or inefficient, because of the effort involved in the implementation. The main drawback of tightly coupled or embedded support systems is that the application semantics now has mixed abstractions. Mixed abstractions lead to software that is difficult to design, implement, validate and maintain.

Our approach to a Change Management Functional Reference Model via abstract machines addresses only the criterial (logical) change management characteristics in section 3, not the environmental (operational) characteristics. That is, only the change management abstraction is covered.

The key challenge in change management is to implement a system that involves minimal overhead and works transparent to the application. Key issues are the granularity of objects which are managed by the system, whether the system can be switched on and off, and whether change management can be inserted into existing applications with minimal effort. Our model is an object-oriented, generic (domain-independent), polymorphic change management system that interfaces seamlessly to applications. Polymorphism refers to the fact that change is managed for all types of objects in a uniform manner.

In the functional model, the change management functions are classified into three orthogonal tasks for managing *Versions*, *Configurations*, and *Transformations*. Each of these tasks is then implemented using an *abstract machine*. The model is illustrated in Figure 1 and the abstract machines are described below.

An abstract machine is a software module which provides an interface to client applications and maintains an internal state; this is analogous to machine instructions and machine state in a hardware machine. The clients communicate with an abstract machine via the interface. The clients have no knowledge of the data structures of the machine and cannot access its internal state. The machine manages a namespace of the client objects that it needs to know about. In our model, all information relating to change management are stored in the internal states of the three machines. The idea of abstract machines as a technique for defining systems and designing software is illustrated in [17] for

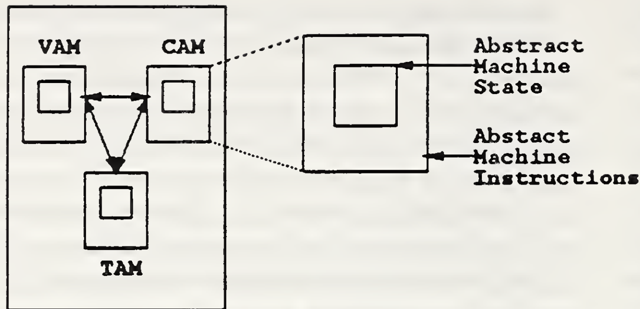


Figure 1: Functional Reference Model

a hypertext system.

The abstract machines approach is necessarily a client server model. The clients, namely the applications, call on the abstract machines to perform change mangement services for the application. Application data structures do not know about the change management information structure. The information is formatted and provided to the application on request. This contrasts sharply with systems where change management structures are part of the application data model. This contrast is illustrated in Figure 2.

We have chosen to model change management as a combination of abstract machines to guarantee that change management is easy to install and use, involves minimal perturbation to application data structures and can live seamlessly on top of applications. The separation of the model into three *orthogonal* machines ensures that an application which needs only one aspect of change management, for example versions but not transformation management, gets only that aspect. Since the model is polymorphic, application programmers see the same interface independent of the domain of the application and independent of the underlying persistent medium, if any.

The reference model has a policy layer which allows it to be customized to implement policies of an organization regarding the management of change. For instance, the policies of one organization might dictate that extensive audit trails be kept; whereas another organization may have no need for audit trails. Policy layer is used to communicate global parameters like number of versions kept and the type of differential versions. Parameters at a finer level of detail can be specified in the interface function calls; this is also the way to override some policy temporarily. The policy layer is a very powerful concept and is another example of the separation of abstractions.

The generic nature of the model implies that it does not know, until told, about application data types. There are some operations like merging and creating differentials which require data type or application specific information. This information is provided, during initialization, using function parameters. These functions, which execute in the application work space, are called by the abstract machines when domain specific actions are to occur.

5.1 Version Abstract Machine (VAM)

VAM provides support for evolution of objects. VAM supports linear and branching histories and merging of histories. The histories may be based on non-temporal or even non-quantitative attributes in the application domain. For example, in a circuit design domain, chips may have histories based on performance as Fast chip, Faster chip etc.. In the same domain, entire designs may evolve based on a confidence level as Completed Design, Validated Design, Tested Design etc. In the usual case of histories based on time, VAM supports timestamps, baselining and major/minor/experimental versions. An *Undo* facility provides for "what if" type of experimentation. VAM provides for facilities to specify how objects are to be merged and how differential versions are to be derived.

5.2 Configuration Abstract Machine (CAM)

CAM provides support for layered or hierarchically structured objects. A configuration is the specifi-

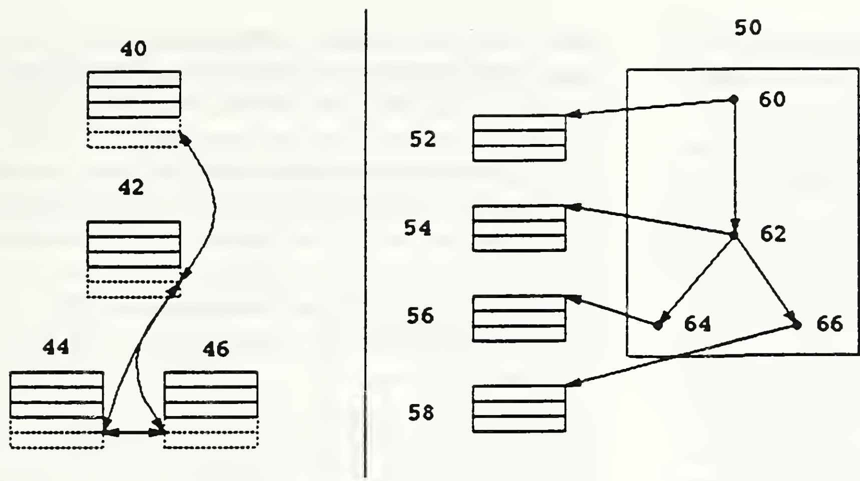


Figure 2: Traditional Model Vs Proposed Reference Model

cation of the composition of an object. CAM supports the composition of objects into graphs. The configurations that CAM supports may be *abstract*, *concrete*, or *dynamic* as defined in Section 2. The binding parameter in a configuration may be versions as supported by VAM. Using such dynamic configurations, an application can specify layered objects which configure themselves as component hierarchies evolve over time or some other attribute. For such dynamic configurations, CAM accesses the Version Abstract Machine through its interface. This is an example of how an application can take advantage of the separation of abstractions provided by the model.

5.3 Transformation Abstract Machine (TAM)

Operations applied to objects during their life cycle are transformations. There are transformations such as editing, simulation, and analysis applied to particular views of an object. There are also transformations, variously called *translations, compilations, expansions, or synthesis*, to bring an object from one view to another. In an object-oriented world, the transformations on an object form part of the behavior of an object; as such, they are the concerns of the designer of the object and not of a change management system. The Transformation Abstract Machine addresses issues like change notification, change propagation, dependency tracking, and constraint maintenance. Object Oriented applications are typified by complex objects which are related to each other and are dependent on each other. The dependencies among objects often reflect on transformations; sometimes they are introduced as a result of transformations. TAM should provide a uniform mechanism to utilize information collected by various application tools in enforcing dependencies. It appears that TAM needs to support a constraint specification and management component and a component for managing consistency between multiple representations or views of an abstraction.

At Texas Instruments, we have prototyped the VAM and CAM abstract machines in a system called Mirage. The prototypes are in Lisp on TI Explorer machines and in C++ on Unix workstations. The exact nature and design of TAM is not fully known at this time. The interface of VAM is listed in Appendix A; the interface of CAM is listed in Appendix B. We have used the Version Abstract Machine to implement change management as a layer on top of the object manager of Lisp Zeitgeist[20].

6 Standards Activity

Some relevant standards work already exists in the change management area. ANSI/IEEE Std 729-1983 (Glossary of Software Engineering Terminology) provides a glossary of over 500 terms including terms in change management. ANSI/IEEE Std 828-1983 (Software Configuration Management Plans) provides a descriptive reference model for configuration management of software items. Other IEEE/ANSI standards in software engineering[25] also relate to some aspects of change management.

X3J13.1 Common Lisp Object System (CLOS) provides basic support for (transient) schema modification and instance evolution.

DoD 2167A documents requirements for managing the entire lifecycle of software designs, from a functional decomposition point of view. As such, it implicitly defines requirements of a change management system, but in addition calls for other kinds of lifecycle support like cost estimation. A key takeaway is that change management is integral, not just to OODBs, but to lifecycle development. In terms of application inter-operability frameworks (like Object Management Group, Engineering Information System, or CAD Framework Initiative), it can be viewed as a service available on the "object bus" or "software backplane".

There is considerable value in standardizing a Change Management model. There is a commonality of requirements and functionality in the task of managing change for various applications. Traditionally, change management systems are custom built and the work in the area of managing change has been embedded into other application sub-systems (databases, file systems, interfaces). There is a growing realization, however, that the time has come for change management to be seen as a primary function[26,30].

7 Conclusion

In this paper, we developed a descriptive reference model by identifying basic characteristics of change management systems. We then surveyed and compared representative existing systems. Change management systems need to support design objects on the appropriate level of granularity, of arbitrary type, with different implementations of the same interface, with multiple data representations, and distributed across machine boundaries. To appropriately build design objects, change management systems need to maintain and enforce the transformations and dependencies on the objects. Since design objects evolve over their lifecycle, change management systems must keep track of the versions and changes of the objects. Designs can be complex; so support of configuration, validation, and multiple users is required. Finally, change management systems need to provide documentation and query capabilities on design objects for the designers and have to be usable systems.

We then presented (two out of three major subsystems of) a precise functional reference model, in the form of a change management abstract machine, which refines the descriptive reference model. The functional reference model describes a domain-nonspecific, generic change management system.

The functional reference model has been partially implemented. While part of the Zeitgeist Open OODB system[44], it is designed to be a separable module. One claim of the model is that it can be used to insert change management into application systems with no perturbance to their data structures. We conducted an experiment to see how much effort is required to insert our change management implementation into an application. The experiment was done in the context of a health care system. The experimental system manages medical records of physicians in private practice. Versioning is used to manage patient histories (linear versioning), consultation records (branching of history), and diagnosis with information from multiple sources/consultations (merging). The insertion took very few lines code; the interface did not change at all except for additional commands in the interface for the user who wanted to query the version information explicitly. Change management is totally transparent to the user except for such added functionality.

References

[1] T. Andrews and C. Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In N. Meyrowitz, editor, *OOPSLA '87 Conference Proceedings*, pages 430–440, ACM, ACM, Orlando, FL, October 1987.

[2] J. Banerjee, W. Kim, H.J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. 1987 ACM-SIGMOD Int. Conf. on Management of Data*, 1987.

[3] D. S. Batory and Won Kim. Modeling Concepts for VLSI CAD Objects. *ACM Transactions on Database Systems*, 10(3):322–346, September 1985.

[4] John F. Beetem. *Structured Design of Electronic Systems using Isomorphic Multiple Representations*. PhD thesis, Stanford University, June 1982.

[5] Bhateja R. and R. H. Katz. A Validation Subsystem of a Version Server for Computer-Aided

Design Data. In *Proc. 24th ACM/IEEE Design Automation Conference*, Miami, FL, June 1987.

[6] Blair R. CM-PATCH (User's Manual). Computer Science Center, Texas Instruments Inc., Dallas, TX, June 1985.

[7] A. Borning. Thinglab - A contraint-oriented simulation laboratory. PhD thesis, Stanford University, 1979.

[8] A. Borning and R. Diusberg. Constraint-based Tools for Building User Interfaces. *ACM Transactions on Graphics*, vol. 5, October 1986.

[9] A. Borning, R. Diusberg, B. Freeman-Benson, A. Kramer, and M. Woolf. Constraint Hierarchies. In *Proc. OOPSLA'87 Conf.*, pp. 48-[illegible] 1987.

[10] Pehong Chen and Michael A. Harrison. *Multiple Representation Document Development.* Technical Report UCB/CSD 87/367, Computer Science Division, UC berkeley, Berkeley, CA, July 1987.

[11] H.T. Chou and W. Kim. A Unifying Framework for Versions in a CAD Environment. In *Proc. 12th VLDB Conference*, pages 336–344, Kyoto, Japan, August 1986.

[12] H.T. Chou and W. Kim. Versions and Change Notification in an Object-Oriented Database System. In *Proc. 25th Design Automation Conference*, pages 275–281, Anaheim, CA, June 1988.

[13] *Introduction to the CLF Environment.* CLF Project, Marina Del Ray, CA, release 1.0 edition, March 1986.

[14] Dan Conde. Bibliography on Version Control and Configuration Management. *ACM SIGSOFT Software Engineering Notes*, 11(3):81–84, July 1986.

[15] W. Courington, J. Feiber, and M. Honda. Network Software Environment Tackles Large Scale Programming Issues. *Sun Technology*, pages 49 - 53, Winter 1988.

[16] C. J. Date. *An Introduction to Database Systems.* Reading, MA: Addison-Wesley, 1986.

[17] N. M. Delisle and M. D. Schwartz. Neptune: A hypertext system for CAD applications. In *Proc. of the International Conference on Mangement of Data*, pages 132–143, Washington, DC, May 1986.

[18] Stuart I. Feldman. Make - A Program for Maintaining Computer Programs. *Software - Practice and Experience*, 9(4):255–265, April 1979.

[19] D. Fishman, D. Beech, H. Cate, E. Chow, T. Connors, J. Davis, N. Derrett, C. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. Neimat, T. Ryan, and M. Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, vol. 5, pp. 48-69, January 1987.

[20] S. Ford, J. Joseph, D. Langworthy, D. Lively, G. Pathak, E. Perez, R. Peterson, D. Sparacin, S. Thatte, D. Wells, and S. Agarwal. Zeitgeist: Database Support for Object-Oriented Programming. In *The Proc. of the Second International Workshop on Object-Oriented Database Systems*, pp. 23-42, 1988.

[21] D. M. Gedye and R. H. Katz. Browsing the Chip Design Database. In *Proc. 25th ACM/IEEE Design Automation Conference*, pages 269–274, Anaheim, CA, June 1988.

[22] I. Goldstein and D. G. Bobrow. Descriptions for a Programming Environment. In *Proc. of 1st Annual Conference on Artificial Intelligence*, pp. 187-189, Stanford, Ca, 1980.

[23] I. Goldstein and D. G. Bobrow. A layered approach to software design. In *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandwall, Eds., New York, NY: McGraw-Hill Book Company, 1984, ch. 19, pp. 387.

[24] Keith Hall. *A Framework for Change Propagation in a Design Database.* Working Paper, Computer Science Department, Stanford University, CA., 1989.

[25] IEEE/ANSI. *Software Engineering Standards.* IEEE, New York, NY., 1987.

[26] John Joseph, Satish Thatte, Craig Thompson, and David Wells. Object-Oriented Database Workshop (OOPSLA '88) In *SIGMOD Record*, September, 1989.

[27] R. H. Katz. *Information Management for Engineering Design.* Springer-Verlag, Berlin, 1985.

[28] R. H. Katz, E. Chang, and R. Bhateja. Version Modeling Concepts for Computer-Aided Design Databases. In *Proc. ACM SIGMOD Conference*, pages 379–386, Washington, DC, May 1986.

[29] Randy H. Katz. *Towards a Unified Framework for Version Modeling.* Technical Report UCB/CSD 88/484, Computer Science Division, UC berkeley, berkeley, CA, December 1988.

[30] William Kent. Panel: an Overview of the Versioning Problem. *Proceedings of SIGMOD Conference,* Portland, 1989.

[31] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrel Woelk. Composite Object Support in an Object-Oriented Database System. In *Proc. OOPSLA'87 Conference*, pages 118–125, Orlando, FL, October 1987.

[32] Gordon S. Landis. Design Evolution and History in an Object-Oriented CAD/CAM Database. In *Proc. 31st IEEE Computer Society International Conference*, pages 297–303, San Francisco, CA, March 1986.

[33] Wm Leler. *Constraint Programming Languages - Their Specification and Generation.* Reading, MA: Addison-Wesley, 1988.

[34] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented Dbms. In *OOPSLA '86 Conference Proceedings*, pages 472–482, ACM, ACM, New York, NY, September 1986.

[35] D. Moon, R. M. Stallman, and D. Weinreb. *Lisp Machine Manual (6th edition).* Cambridge, MA: MIT, 1984.

[36] Alan Purdy, Bruce Schuchardt, and David Maier. Integrating an Object-Server with Other Worlds. *ACM Transactions on Office Information Systems*, 5(1):27–47, January 1987.

[37] Rice, J. *Source Code Control System (User's Manual).* Stanford University, Stanford, CA.

[38] M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[39] J. Rumbaugh. Controlling Propagation of Operations Using Attributes on Relations. In *Proc. OOPSLA'88 Conference*, pages 285–296, San Diego, CA, September 1988.

[40] A. Skarra and S. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *OOPSLA '86 Conf. Proc.*, pp. 483 - 495, 1986.

[41] D. Smith, G. Kottik, and S. Westfold. Research on Knowledge-Based Software Environments at Krestel Institute. *IEEE Transactions on Software Engineering*, November 1985.

[42] G. L. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* MIT Tech. Rep. AI-TR.595, 1980.

[43] Walter F. Tichy. Design, Implementation, and Evaluation of a Revision Control System. In *Proc. 6th International Conference on Software Engineering*, page , Tokyo, Japan, September 1982.

[44] Craig Thompson, David Wells et. al. Open Architecture for Object-Oriented Database Systems. Computer Science Center, Texas Instruments Inc., Dallas, TX, Dec. 1989.

[45] R Waters. *KBEmacs: A Step Toward the Programmer's Apprentice.* Technical Report 753, Massachusetts Institute of technology, Boston, MA, May 1985.

[46] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, pp. 63-75, March 1990.

Appendix A: Version Abstract Machine

A.1 Purpose of Module

The Version Abstract Machine allows the clients to create and manage linear and non-linear versions of objects, query the history of objects, support dynamic configuration of objects and merge versions of objects according to user supplied criteria. In a typical application, the application programmer would have inserted the server requests to the machine in the proper places so that versioning takes place transparent to the application user. The application programmer would provide version accessors written in the application's terminology to the user.

A.2 Design/Implementation

For each object that is versioned, the Version Abstract Machine (VAM) manages a graph of version nodes. Each version node contains information that VAM needs to manage versioning and a pointer to (or an id of) the object that the node versions. In the presence of branching versions and merging, this graph is a rooted directed acyclic graph. In this graph, if A is directed to B, we will refer to A as the (immediate) parent of B and B as the (immediate) child of A. A node in the graph may have one or more children (versions). One of these is distinguished as the primary version. When nodes are merged, the merged node becomes a child of each of the nodes that was merged. VAM knows how to update the graph for merging two nodes together; but it has no knowledge about how user objects can be merged to obtain the object pointed to by the merged node. The user gives this information to VAM as described in the functional reference model (section 5).

Each version node has two timestamps associated with it: the actual time the node was created (system time) and a version time specified by the user. If the user does not specify a time, the version time defaults to the system time. Time-stamps are resolved to exact match, if any, or closest match before.

Application programs interface to the Version Abstract Machine Module using Lisp or C functions with arguments of primitive types. These functions are described below.

Interface

This section describes the functions available to an application programmer who wishes to use the Version Abstract Machine (VAM) module. The functions arguments are of type integer, char *, or Boolean. We refer to the object that a version node points to as its value.

VAM has a context called current_graph and each graph has a context called current_node. Many of the functions below take an argument called *Gname* which is a character string. If this is optional and is not specified VAM uses the graph which is known to it as the current_graph. Similarly, if an optional Vname is defaulted, VAM uses current_node. Current_node and current_graph are reset by various VAM operations. They can also be reset by appropriate functions described below. You can query for the current settings of current_node and current_graph also by using functions provided below.

When an interface function returns an aggregate date structure (lists or arrays), the items in the structure are ordered chronologically.

Table 3 lists the interface functions with their arguments and return values. Brief descriptions of the functions are given below. Functions which return a pointer value (char *, char **) should be checked for NULL before using the returned value. Functions which Return TRUE or FALSE should be checked for TRUE (success) or FALSE (failure).

Create Version Graph creates and returns a version graph. This is the first call made to Version Abstract machine to start versioning an object.

Create Version Node creates and returns a version node.

Install Version installs a version node into an existing version graph.

Delete Version deletes an existing version node.

Merge Versions merges two version nodes. The merging of the objects that the nodes address is done by the application. THe merged node is installed as a version of the two nodes from which it is merged.

Set Current Version sets the current version.

Set Version Attribute sets an attribute of the version. Typically this attribute is a user specified time stamp.

Set Current Graph sets the current graph.

Return Root Version returns the root version.

Versions With Value returns all the versions with a given value.

Return Current Graph returns the current graph.

Return Current Version returns the current version.

Describe Node And Value describes the version node and its value in some form specified by the user.

Describe Version Graph describes the version graph basically as a mathematical graph.

Describe Version Node describes the version node by listing parents, children, and siblings with no indication of the node's value.

All Your Versions gives a linear list of all versions of a node.

Find Versions By Attribute finds all version nodes with a given value for a certain attribute. Typically used to find all versions with a certain time stamp.

Version Time returns the time that the user associated with this node, if any. If there is no user specified time attribute, a system time is returned.

Return Child Versions returns all immediate children of a node.

Return Parent Versions returns all immediate parents of a node.

Return Primary Version returns the primary version of a node.

There Exists Version returns true or false as appropriate.

List All Version Graphs returns a list of all the version graphs known to the Version Abstract Machine. Note that there is one graph for each object being versioned.

Browse Version Graph browses a version graph using a browser known to Version Abstract Machine. It is expected that there is a graph browsing facility available.

Return Version Object returns the value of a version node.

Undo Last undoes the last operation. There is no *Redo* in this prototype.

Export To writes out a linearized version of the machine or a graph that is suitable for writing to a disk. In the prototype this is an array.

Import From is the opposite of the above.

Appendix B: Configuration Abstract Machine

B.1 Purpose of Module

The Configuration Abstract Machine allows clients to create and manage compositions of objects, abstract configurations, dynamic configurations of objects that are versioned and concrete configurations of specific object instances. In a typical application, the application programmer would have inserted the server requests to the machine in the proper places so that configuration management takes place transparent to the application user. The application programmer would provide configuration accessors written in the application's terminology to the user.

B.2 Design/Implementation

For each object that is configured, the Configuration Abstract Machine (CAM) manages a graph of configuration nodes. Each configuration node contains information that CAM needs to manage the configuration and a pointer to (or an identifier of) the object that the node configures. The object pointer is weakly typed since an object can be composed of a heterogeneous collection of other objects. In the presence of the sharing of subcomponents among objects, this graph is a rooted directed acyclic graph for each component. In this graph, if A is directed to B, we will refer to A as the (immediate) supercomponent or parent of B and B as the (immediate) subcomponent or child of A. Each node in the graph may by thought of as a root of a configuration graph.

An object pointer in the configuration node can also point to a version graph managed by the Version Abstract Machine (VAM). In this situation, any version of an object associated with the version graph can act as the object associated with the configuration node. A *concrete configuration* is a configuration in which specific versions of objects have

been identified for each configuration node pointing to an object version graph. A *dynamic configuration* is a configuration in which specific versions of objects have only been identified for some of the configuration nodes pointing to object version graphs. An *abstract configuration* is a configuration in which only the structure of the composition has been defined. For example, a book is composed of chapters.

Application programs interface to the Configuration Abstract Machine Module using Lisp or C functions with arguments of primitive types. These functions are described below.

B.3 Interface

This section describes the functions available to an application programmer who wishes to use the Configuration Abstract Machine (CAM) module. The function arguments are of type object *, char *, or Boolean. We refer to the object that a configuration node points as its value.

Many of the functions below take arguments called *Node, Root-Node* or *Parent-Node* which are character strings.

When an interface function returns an aggregate date structure (lists or arrays), the items in the structure are ordered in the same order that they were inserted.

Table 4 lists the interface functions with their arguments and return values. Brief descriptions of the functions are given below. Functions which return a pointer value (char *, char **) should be checked for NULL before using the returned value. Functions which Return TRUE or FALSE should be checked for TRUE (success) or FALSE (failure).

Create Configuration Node creates and returns a configuration node.

Install Configuration Nodes installs a set of configuration nodes as the subcomponents of a supercomponent configuration node.

Delete Configuration Node deletes the configuration graph rooted at the specified configuration node from its supercomponent configuration nodes.

Undo Last Operation undoes the last operation. There is no *Redo* in this prototype.

Return Children of Node returns the current set of configuration nodes that are the subcomponents of the current configuration node.

Return Parents of Node returns the current set of configuration nodes that are the supercomponents of the current configuration node.

Return Configuration Graph returns a new configuration node rooted at the same configuration graph as the current configuration node but whose subcomponents configuration nodes are determined by a specification function.

Describe Configuration Graph describes the configuration graph rooted at the current configuration node.

Find Configuration Graph by Description finds all configuration nodes rooted at configuration graphs whose descriptions match the given description.

Browse Configuration Graph browses the configuration graph rooted at the current configuration node using a browser known to the Configuration Abstract Machine. It is expected that there is a graph browsing facility available.

Export To writes out a linearized version of the machine or of a graph that is suitable for writing to a disk. In the prototype this is an array.

Import From is the opposite of the above.

Function Name	*Required Args*	*Optional Args*	*Return Value*
Create Version Node	Object, Gname	Vname	Vname
Create Version Graph	Gname	Nil	Gname
Install Version	Vname, Gname	Parent, Primary-p	Boolean
Delete Version	Vname	Gname	Boolean
Merge Versions	Vname1, Vname2	Gname, Merged-Name	Merged-Name
Set Current Version	Vname	Gname	Boolean
Set Version Attribute	String, Vname	Gname	Boolean
Set Current Graph	Gname	Nil	Boolean
Return Root Version	Nil	Gname	Vname
Versions With Value	Value	Gname	List-Of-Names
Return Current Graph	Nil	Nil	Gname
Return Current Version	Nil	Gname	Vname
Describe Node And Value	Vname	Gname	Void
Describe Version Graph	Nil	Gname	Void
Describe Version Node	Vname	Gname	Void
All Your Versions	Vname	Gname	List-Of-Vnames
Find Versions By Attribute	String	Gname	List-Of-Vnames
Version Time	Vname	Gname	Time-String
Return Child Versions	Vname	Gname	List-Of-Vnames
Return Parent Versions	Vname	Gname	List-Of-Vnames
Return Primary Version	Vname	Gname	Vname
There Exists Version	Vname	Gname	Boolean
List All Version Graphs	Nil	Nil	List-Of-Graphs
Browse Version Graph	Gname	Nil	Void
Return Version Object	Vname	Gname	Object
Undo Last	Nil	Nil	Void
Export To	Version-Machine	Nil	External-Form
Import From	External-Form	Nil	Version-Machine

Table 3: Interface to Version Abstract Machine

ction Name	*Required Args*	*Optional Args*	*Return Value*
ate Configuration Node	Object, Timestring	Cname	Node
all Configuration Nodes	List-Of-Nodes, Parent-Node	Nil	Graph
ete Configuration Node	Node, Parent-Node	Nil	Graph
lo Last Operation	Nil	Nil	Void
urn Children Of Node	Node	Nil	List-Of-Nodes
urn Parents Of Node	Node	Graph	List-Of-Nodes
urn Configuration Graph	Node, Node-Function	Nil	Graph
cribe Configuration Graph	Graph	Nil	Void
d Configuration Graph By Description	String	Nil	List-Of-Nodes
wse Configuration Graph	Graph	Nil	Void
ort To	Configuration-Machine	Nil	External-Form
ort From	External-Form	Nil	Configuration-Machine

Table 4: Interface to Configuration Abstract Machine

EIS/XAIT Project: An Object-based Interoperability Framework for Heterogeneous Systems[1]

(Position Paper)

Girish Pathak, Bill Stackhouse, and Sandra Heiler

Xerox Advanced Information Technology

Cambridge, Massachusetts

Abstract

This paper briefly describes various technical issues involved in the design and development of an object-based interoperability framework in support of Engineering Information Systems (EIS). It also discusses the interaction of such frameworks with various emerging standards and the possibility of developing standards in the area of object-oriented interoperable frameworks. Finally, it summarizes on the background and the status of the project.

1. Introduction

Significant investments in existing software systems or tools (some of which may be a hardware specific), emergence of large software systems, continued advances in powerful yet user-friendly languages, and prohibitive cost of building software systems from scratch, have led to the

[1] This work was supported by the DoD under contract F3615-87-C-1407. For further information, contact Girish Pathak at pathak@xait.xerox.com on ARPAnet or (617) 499-4498.

research and development of interoperability frameworks or "software infrastructures" as Bershad, etc [Bershad87] refer to it. The objective of these software frameworks is to decrease the marginal cost of adding a new type of system to an existing computing environment, and at the same time to increase the set of common services that users would like to share.

A number of research projects share these objectives. The Mixed Language Programming System (MLP) [Hayes87], at the University of Arizona at Tuscon, facilitates construction of programs in which procedures can be written in different programming languages. The MLP system is built entirely above the operating system and consists of a runtime system, a translator for each host language (currently C, Pascal, and Icon) and a linker. Cross language invocations are done by message passing. Similarly, the Agora project [Bisiani88] at Carnegie-Mellon University supports the development of multi-language parallel applications on heterogeneous machines.

However, there are two major differences between the objectives of these efforts and that of ours: the need to support abstraction and the need to manage non-persistent as well as persistent objects. Our application, the engineering design and development environments, involves arbitrarily complex data types. And these environments must support long and interdependent tasks that create and manipulate specialized data. Often multiple representations of the same information are required to support different tasks. Therefore, the notion of abstraction and the management of information becomes a necessity in such environments.

The work described in this paper is the result of a project that is addressing the requirements of [Linn86] for evolving Engineering Information Systems (EIS). These requirements can be briefly stated as follows:

- make arbitrary tools and databases interoperable,

- avoid requiring changes to underlying systems and repositories,
- enforce management controls,
- provide tailorability,
- ensure compatibility among EISs, and between EISs and non-EIS systems, and
- provide a uniform environment for tool builders.

2. The Object-Based Interoperability Framework

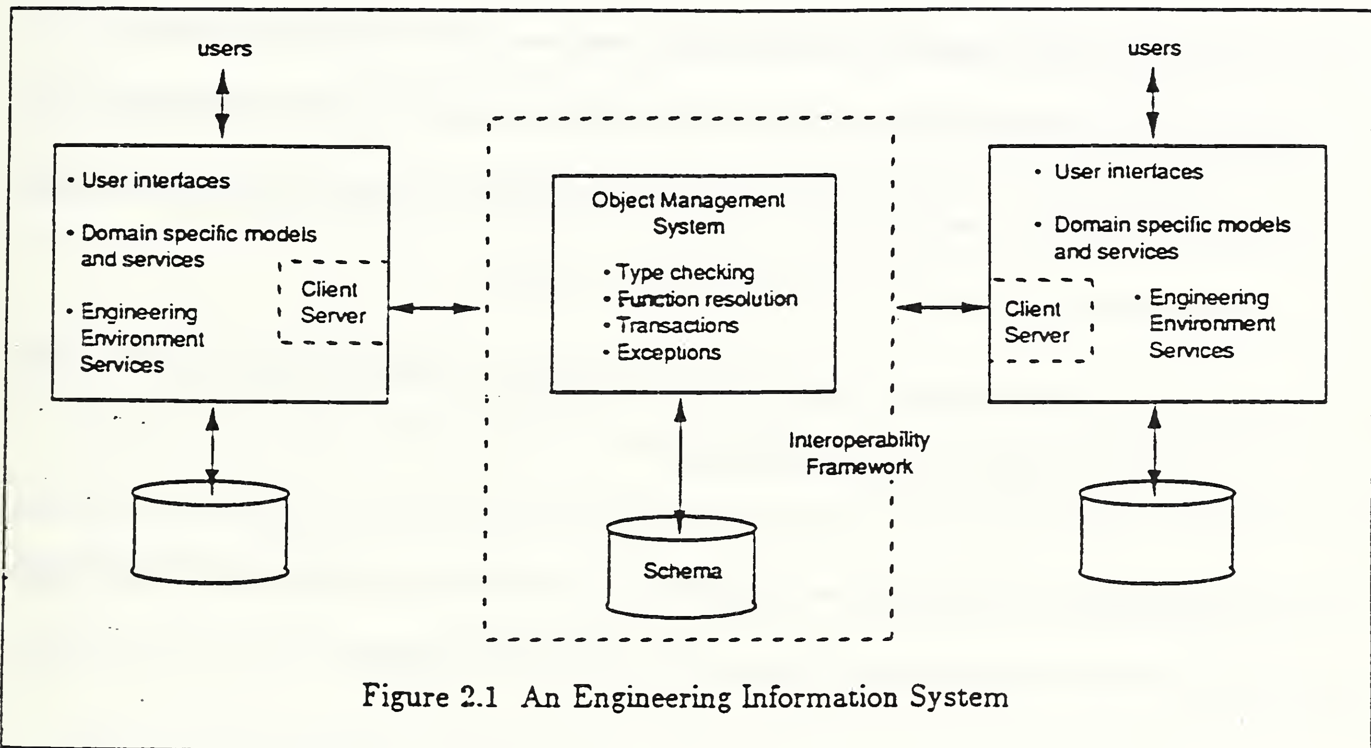


Figure 2.1 An Engineering Information System

In response to the requirements of engineering information systems described in [Linn86], Xerox Advanced Information Technology (XAIT) has designed an object-based interoperability framework, hereafter referred as Object Management System (OMS). The relationship of OMS with rest of the components of an EIS is shown in Figure 1. The benefits of using an object model include (but are not limited to):

- a richer data model which closely matches applications,
- typing and inheritance facilitaties that promote reuse of software components,
- data abstraction and encapsulation,
- facilitating modular development,
- simplified management of policies and protocols,
- allowing different levels of integration, etc.

The object model of OMS provides the following facilities to enable interoperability:

1. It provides built-in abstract data types and constructors for user-defined abstract data types that allow their components to be described in ways that hide their heterogeneity.
2. It provides reference resolution, type checking, and function resolution mechanism to reference and operate over objects from heterogeneous systems.
3. It provides various execution related primitives including transaction management support and exception handling for invoking function.

In the following two subsections, we describe how a combination of three primitives and a function applier facilitate interoperability among heterogeneous EIS systems.

2.1 Primitives

The object model is based on three primitives: object, function, and type. An object is an abstraction (structure and behavior) with an identity which is independent of its value or relationship to any other entity in the system. The identity of an object is captured by a unique and immutable identifier referred to as the OID. Each object is associated with a type that describes its structure and behavior. Everything, including functions and types, is defined as an object in the model. Integers, employees, designs, network nodes, bytes, and string are all objects.

A function is a mapping from some input objects to output objects. Functions are applied to an object to yield its properties (attributes) or related objects, to perform operations on it, or to test constraints on it. For example, one might apply the name function to an employee object to yield the string object that represents the employee's name; or one might apply the department function to the employee object to yield the department object that represents the department where the employee works. Functions have signatures which specify the name of the function, type of its input argument, types of output arguments, and pre- and post-conditions that must be true before and after function execution.

A type is a description of the structure and behavior of some object(s) -- instances of object. The type specifies the set of functions that can be applied to any instances to yield that object's properties or related objects, or to perform operations on the object, or to test constraints on it.

2.2 Function Application

In the object model, interoperability is achieved through function application. The body of a function is a procedure written in some programming language which implements the semantics of the function. A request to apply a function to argument objects involves invoking this procedure by passing parameters through object references and receiving results through object references.

The function application includes: type checking, resolution of function and references, invocation of an associated procedure that implements the function, and claiming results. The model ensures strong typing which provides needed assurance for large complex integrated systems.

An implementation of the typeCheck function can be provided for each type. This allows specific tests of objects to be made to determine whether they meet the requirements of specific components for that type. For example, the typeCheck function implementation provided for type integer might test the representation of the object to see whether it is a valid bit pattern for integers on the particular machine where the test is made, whereas, the typeCheck function imple-

mentation for type employee might do a database query to determine whether the referenced employee is listed in the employee roster. This is important where the components of the federation are autonomous and objects can be deleted or created and assigned to types outside the knowledge of the federated system.

Resolution involves selecting an implementation (body) of the function for invocation. It is provided by a resolution function applied to the function application.request. It includes mapping from the abstract argument specified in the request to the representation used by the code of the implementation and passed as a parameter to it. The body can be provided to the type through inheritance (or delegation) as well as provided specifically for the type of the argument. Notice that archetypes participate in isA hierarchies, through which functions and their implementations are inherited, as well as other types and metatypes.

Invocation of the procedures that provide function implementations is provided by machine/operating system/programming language-specific "execution engines," which are implemented as functions on implementation object types, e.g., the UnixShell Program. An invoked procedure returns outputs by applying a postOutputs function, which allows objects to be claimed by the system and returned to the client/user.

The posting of outputs and the raising of exceptions are performed exactly like any other operations, by applying functions. Where the invoked procedure consists of existing code that cannot be modified, the request to map the argument object to its representation or to post results or raise an exception are performed in "adapters" or wrappers around the code that handles requests specific to the federated environment.

3. Transaction Management

The transaction management facilities for the engineering design and development environment must support transaction primitives whose semantics are implemented by underlying autonomous heterogeneous systems, which can be extended to support cooperation, and which provide support for long running activities.

The OMS supports nested transaction as described in [Moss86]. Nesting of transactions provides both greater concurrency (by allowing more tasks to run in parallel) and limits the recovery overheads to a minimum.

The correctness criteria is facilitated through the concurrency protocols among transactions. The correctness criteria of strict serializability may be too restrictive for such environments. Therefore, each set of nested transactions is designed to adopt a concurrency protocol which is weaker than its parents and thereby offers the potential for increased concurrency and cooperation among transactions [Skarra89].

Transaction management functions are implemented through mappings to requests to the underlying data servers. For example, two-phase commit is provided for transactions in the federation only where all the component systems that store data updated by the transaction provide two-phase commit facilities.

4. Views and Query Language

The purpose of the view mechanism is to provide information hiding, access control, and hiding heterogeneity. A view consists of a set of objects and a set of types which are bound [Heiler90]. That is,

V = [{o: object}, {t: type}] where type(o) is some t

The view provides an object base and a schema that determines the context in which a user program applies its functions. Type checking and resolution are provided in the context of a view in the OMS system.

The query language consist of the functions that operate on collection types. They take instances of specific set types as input arguments and produce set types as output. The functions are provided on the metatype set and specialized for the specific types through parameterization.

The specific set of functions include:

- union, intersection, and difference operators,
- select, project, and join, and
- functions for flattening and grouping sets with embedded structures.

The query language provides a uniform set of operations across heterogeneous data stores. With this uniform set of operations, optimization can be performed to reduce data transformations and transfers across a distributed heterogeneous system. It is envisioned that standard methods would be provided to map the query operators onto OODBMS, relations DBMS, and various standard file systems.The query operators are also used by the OMS to derive views and types within the system data.

5. Standards and the Interoperability Frameworks

Many individual standards such as IGES/PDES, EDIF, VHDL have attempted to standardize on portions of an application domain's collection of problems. Recently, standard activities such as EIS and CFI (CAD Framework Initiative) have begun to address issues that relate to the full lifecycle of a particular application domain such as electronic design automation.

Many contemporary engineering activities include some degree of electrical, mechanical, and software engineering. A single framework could be introduced to allow the integration of all activities. It is possible that the framework could be extended to other application domains such as banking, insurance, etc. A general framework which could accommodate the integration of all tools used during the life-cycle would:

- open up new markets,
- allow tool builders to have specialized products with a wider audience,
- simplify tool development, and
- give customers control over the storage and access of their data.

Once the general framework is defined, domain specific (electronic, mechanical, software, etc.) standards can be introduced for representation of information.

6. The Project Background and Status

The OMS is a general framework designed to work in any application domain. The specification of the OMS began in mid-1987 as part of the EIS contract. In late 1989, the specifications were completed. In 1990, a prototype OMS is being developed to demonstrate the feasibility of such a framework.

7. References

[Bershad87] B. Bershad, et. al. "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems," IEEE Transactions on Software Engineering, Vol. SE-13, No. 8, pp. 880-894, August 1987.

[Bisiani88] R. Bisiani. "Multilingual Parallel Programming of Heterogeneous Machines," IEEE Transactions on Computers, Vol. 37, No. 8, pp. 930-945, August 1988.

[Hayes87] R. Hayes and R. Schlichting. "Facilitating Mixed Language Programming in Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-13, No. 12, pp. 1254-1264, December 1987.

[Heiler90] S, Heiler and S. Zdonik. "Object Views: Extending the vision," Proceedings of International Conference on Data Engineering, IEEE, 1990.

[Linn86] J. Linn and R. Winner. "Department of Defense Requirements for Engineering Information Systems," Institute of Defense Analyses, 1986.

[Moss86] J. E. Moss, "An Introduction to Nested Transactions," COINS Technical Report 86-41, Department of Computer and Information Science, University of Massachusetts at Amherst, 1986.

[Notkin87] D. Notkin, et. al. "Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity," Communications ACM, Vol. 30, No.2, PP. 132-140, February 1987.

[Skarra89] A. Skarra, "Concurrency Control for Cooperating Transactions in an Object-Oriented Database," SIGPLAN Notices, 1989.

Goals and Requirements
Storage Manager (SM) Working Group
Design Data Management TSC
CFI

edited by Drew Wade, drew@objy.com

V0.4
Draft Proposal

ABSTRACT

The Storage Manager Working Group is to specify the standard interface (SMI) to the CFI framework component Storage Manager (SM). The SM is a general-purpose mechanism to provide access to data entities and their relationships, to support the Design Representation component, the Design Data Management component, the remainder of the CFI's framework components, and its applications. This will allow interchangeability of compliant SM's, with minimal effect on the application.

CONTENTS

1. What is SM?

2. Goal of SM

3. Relation to CFI Goals

4. SM Design Considerations

5. SMI Guiding Principles

6. Functionality Areas

7. SM Documents

8. References

9. History of This Document

1. What is an SM?

SM is the Storage Manager component of the CAD Framework. It helps store, manipulate, and manage data for the framework and its applications. The SM Working Group, which is specifying the SM *interface*, is part of the CAD Framework Initiative (CFI), whose structure we now briefly describe.

The CAD Framework Initiative (CFI) is a broad-based industry organization, whose members include users and vendors of CAD applications, as well as vendors of products used in CAD systems. Its purpose is to develop, for design automation frameworks, industry-acceptable guidelines that will enable the coexistence and cooperation of a variety of tools. The CFI has formed several Technical Subcommittees (TSC's), including the Design Data Management (DDM) TSC, whose task is to specify interface standards to support management of design data. The DDM has created five working groups to pursue that task in detail: Data Sharing, Consistency, Versioning and Configuration Management, Types and Schema, and Storage Management. The Storage Management group is producing this document.

The following table lists the Technical Subcommittees of the CFI, along with their subjects:

Architecture	Identifies the framework components and their interrelationships, and addresses global issues.
Design Data Management (DDM)	Design data support.
Design Representation (DR)	Addresses the standard data models.
Design Methodology Management (DMM)	Addresses the services to control the design process.
System Environment (SE)	Identifies the base operating system features.
User Interface (UI)	Addresses issues of consistency in behavior and components to support them.
Intertool Communication (ITC)	Addresses the ability to allow applications to exchange information.

These groups are all specifying interfaces that will allow interchangeability and interoperability of applications, frameworks, and framework components.

Many, if not all, framework components require storage management services. After discussions within DDM and with representatives of other TSCs, notably the Architecture and Design Representation TSCs, we conclude that a single SM framework component could serve the storage management needs of other framework components, as well as of applications. Although not all components or applications in any particular implementation would necessarily use this SM component, they could use it when needed, through the standard interface. This arrangement would save work by implementors of applications and framework components, reduce the size of the resulting system, and enable much

greater integration of tools and functionality, thanks to the sharing of design data in a single storage system.

Therefore, the SM Working Group has undertaken to specify the CFI standard Storage Manager Interface (SMI). It is important to understand that the SM Working Group is *not* attempting to design an SM! Instead, we are specifying an interface that will be standard, so that different SM's can be used by frameworks and applications.

2. Goal of SM

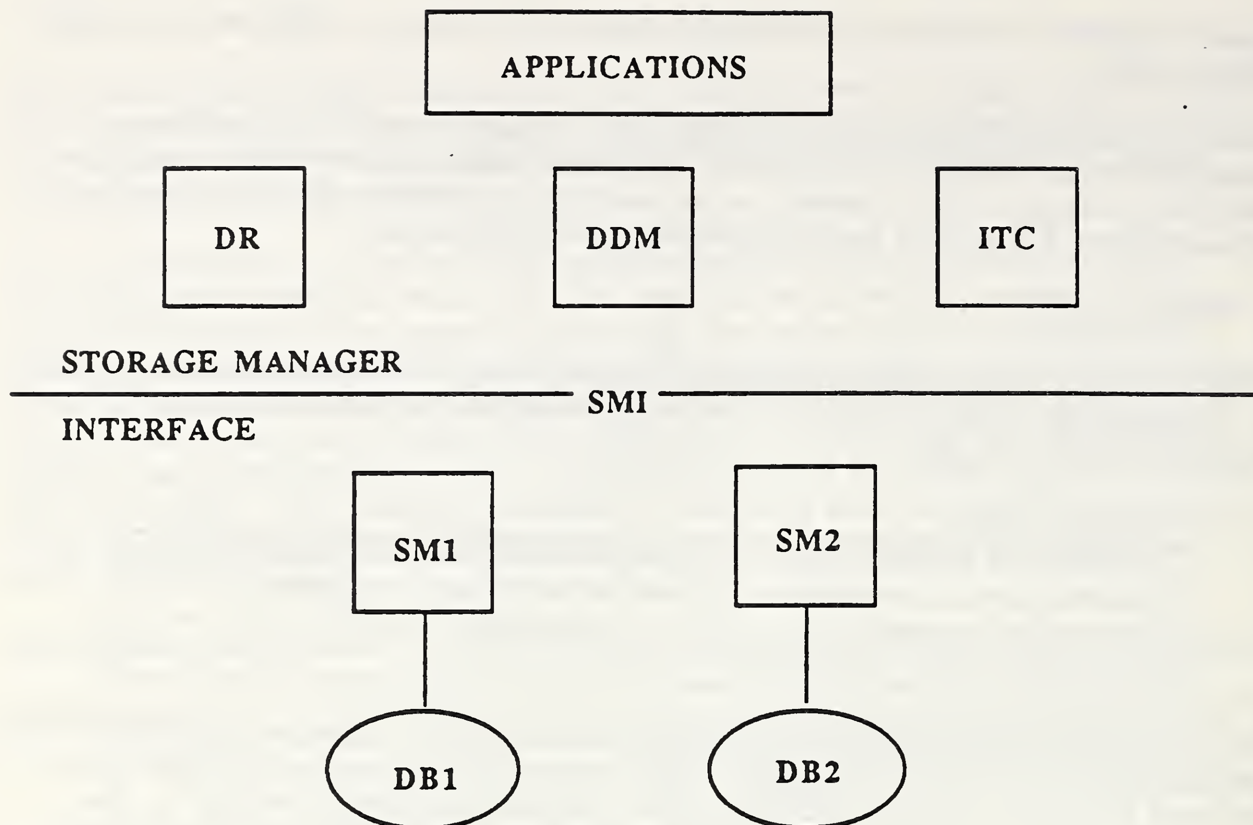
We first state our goal, and then discuss its concepts at length. In later sections we expand on its relationship to the CFI goals, and on the requirements that must be followed in proceeding to meet the goal.

The SM Goal:

To specify the interface to the CFI framework component Storage Manager, a general-purpose mechanism to provide access to data entities and relationships, to support DR, DDM, the remainder of the CFI's framework components, and its applications.

The primary CFI goal from which we take our charter specifies "interoperability and interchangeability of tools and databases." We interpret this to mean interchangeability of databases along with their managers (software management systems) (see Fig. 1). The alternative would be to specify the format of the database (bits on disk) itself, so that any SM software could access that database. Although this would provide a greater level of flexibility, allowing interchange of SM code on the same database image, it would greatly restrict the freedom of the SM implementors, stifling progress in that area. Further, it seems sufficient for the needs of CAD systems and users to be able to interchange the SM along with the data, as long as there is also some mechanism for communicating between SM's and moving data from one to another.

In any layered architecture whenever a lower layer is accessible from a higher layer, the middle layer must define the restrictions on the higher layer's use of the lower layer. For example, if the SM is implemented on top of a UNIX filesystem, the application certainly can still directly access the UNIX interfaces, including fwrite. However, the application may not invoke fwrite directly on the files used by SM since that would conflict with the functioning of SM. Similarly, if the DDM is implemented on top of the SM, the application may still use the SM interfaces, including SM versioning and SM checkout. However, the application may not invoke SM versioning or SM checkout on design objects whose versioning and configuration management is controlled by the DDM.



"INTEROPERABILITY OF DATABASES"
= "INTEROPERABILITY OF SM +DB"
= STANDARD SM INTERFACE = SMI
Figure 1

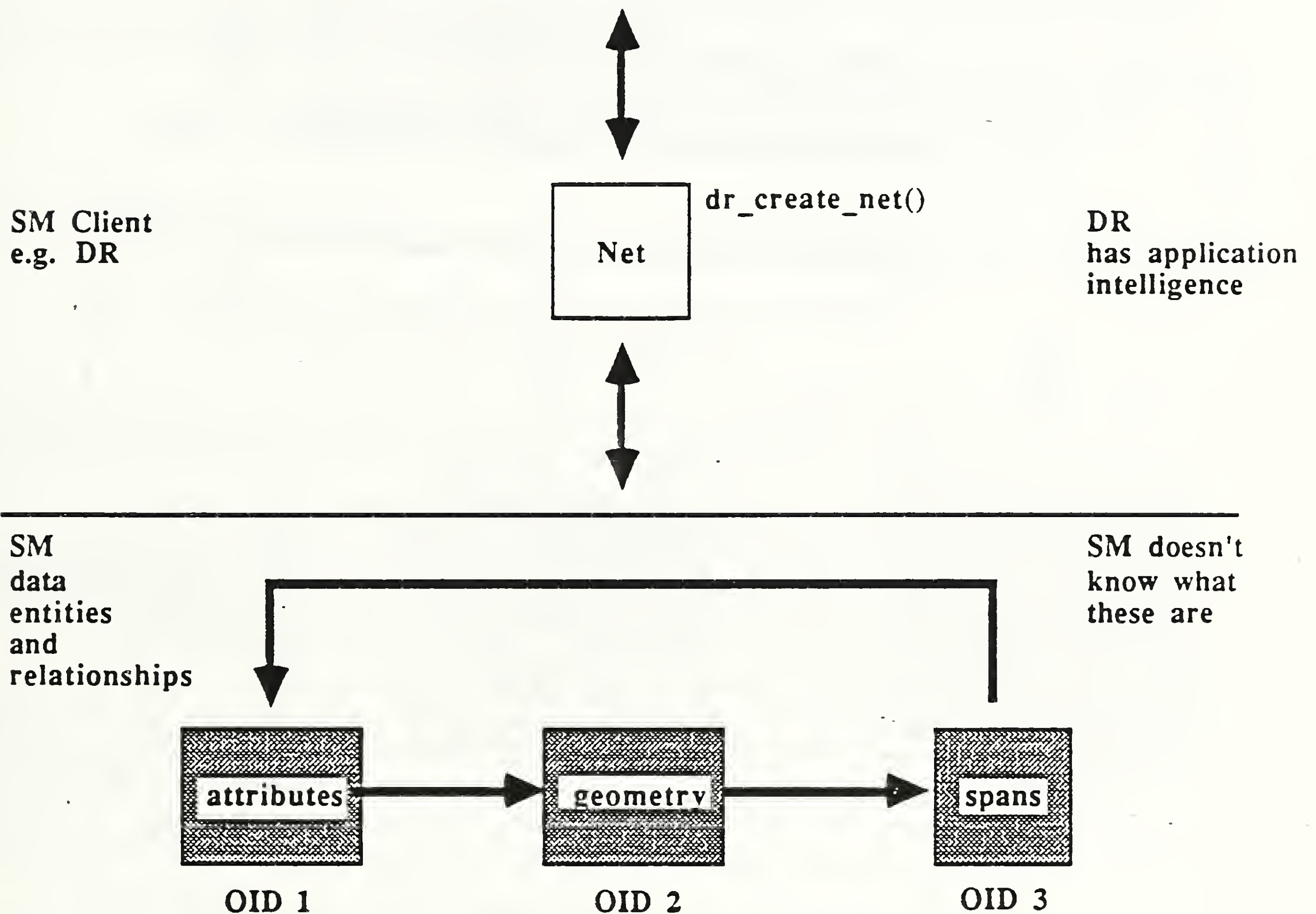
Further requirements must also be met for the SM to be successful, and these must be supported through the SMI. They include support for large designs, large groups of designers, high performance, etc., and are discussed in more detail in the following sections.

The SM Goal also states that the SM is a general-purpose mechanism. It should support storage management needs without being specific to a particular application or application domain. In this way it can serve the needs of many such applications and domains, as well as the framework components. The SM users can freely employ the SM's capabilities to accomplish the appropriate functionality, to model the appropriate constructs, etc. Examples of such uses for specific application domains might include: signals or schematic pages or simulator structures or mechanical assemblies (as in DR); process status, tracking, and dependency structures (as in DMM); menus and graphics and buttons (as in UI); inter-application communication structures and protocols (as in ITC); etc.

In order to achieve this generality and yet supply the necessary capabilities, we have chosen a model of data entities and relationships. A data entity represents a unit of data addressable and accessible by the SM user. It might look like a record (as in traditional DBMS's), a structure (as in C), an object (with associated methods in an object-oriented or OODBMS), or some other format. It might be fine grained (as a net or component or gate), or it might be large grained. Although we use the term *object* for these data entities, and *object identifier* (OID) for a means to uniquely reference them, we do not mean to require object-oriented implementations; rather, we focus on the needs of the framework and the

applications and leave the choice of technology and implementation to the implementor of the SM. Relationships between such data entities are provided by the SM as a modelling tool. They might be used to represent dependencies, references, connections, and composed parts of a larger logical application entity. Note that we specify an interface in terms of these (and related) concepts, rather than in terms of specific physical structures such as files.

We specifically exclude the application-level interpretation of such objects and their relationships from the domain of the SM. It has no understanding of the application itself, but rather provides general-purpose mechanisms that the application can use. An example of this is illustrated in Fig. 2. The mention of OID (or object ID) is meant to specify a unique means of identifying the data entity.

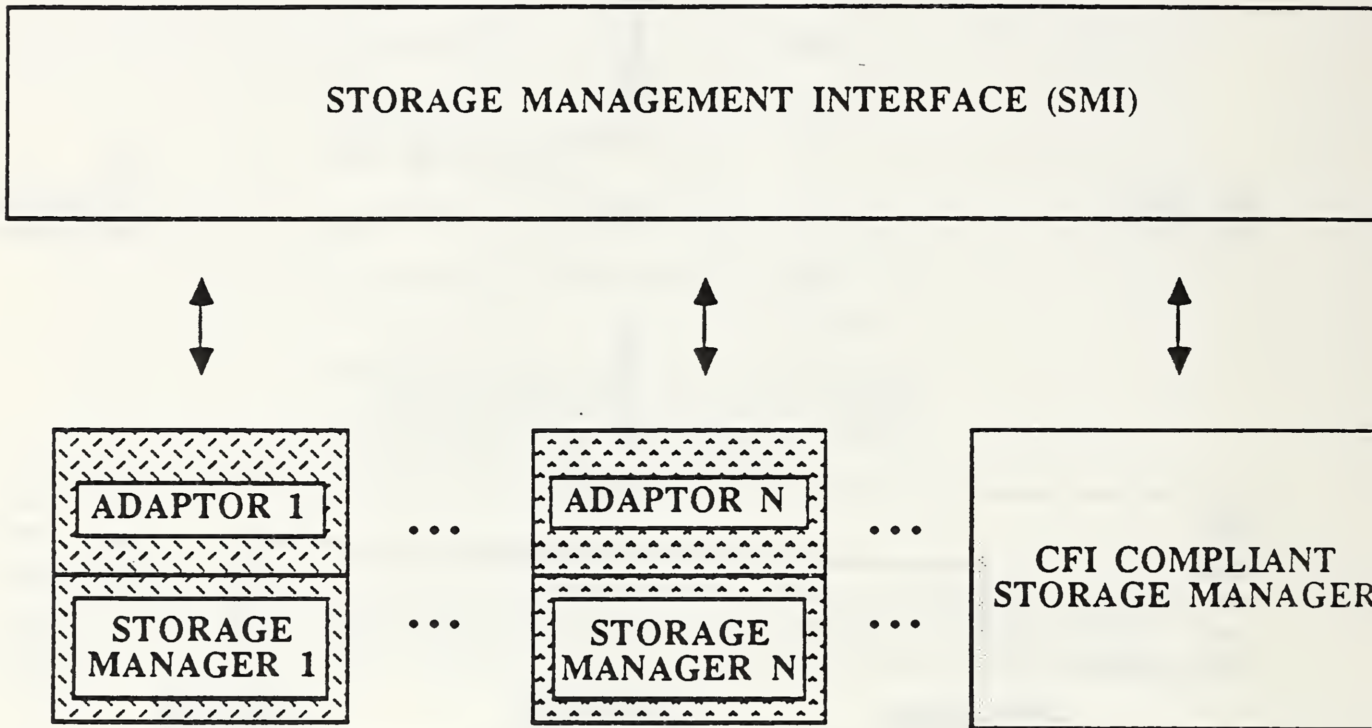


Example of SM's Use within CFI
Figure 2

It is presumed that the full issue of resolution of names within and across frameworks is the province of the DDM. Its name resolver will supply fully, or appropriately, qualified names to the SM. By treating different SMs as different namespaces or scopes, the DDM could support multiple SMs simultaneously.

It is specifically not the province of the SM group to define the architecture, internals, or implementation of the SM itself. It is hoped that multiple implementations will appear that provide CFI compatibility by meeting the SMI interface. As illustrated in Fig. 3, this might be achieved by adding an *adaptor* which might be as simple as a macro to a pre-existing storage manager or DBMS in order to match it to the SMI, or by directly implementing a

new storage manager to meet the SMI interface. It is not our intent to constrain SM's to any particular technology. The storage managers in Fig. 3 might use any of various technologies, including Codasyl or Network, Hierarchical, Relational, Object-Oriented, or custom. As viewed through the SMI, these all appear the same. Our primary focus in defining the SMI is to meet the needs of the framework components and the applications. We wish to encourage wide use of the SMI, so we endeavor to keep existing data managers in mind. However, we do not want to constrain ourselves to such existing systems at the cost of failing to meet the needs of the framework and applications.



SMI Support via Different Storage Managers

Figure 3

3. Relation to the CFI Goals

The primary CFI goal from the outset has been to provide standards that would support "interoperability of tools and databases." As stated before, we interpret this to mean interchangeability of data along with its manager, as in Fig. 1.

The most recent version of the CFI goals document[3] specifies seven goals, each containing several objectives. Two of these goals directly lead to our charter (among others), and two others impose requirements on us.

The two with direct impact and their relevant objectives are:

G1: Facilitate design in the large
- O1.1 Support multiple users and design teams
- O1.2 Support a distributed computing environment
- O1.3 Support a heterogeneous computing environment

G3: Management, sharing, reuse, and exchange of engineering information
- O3.2 Data interchangeability with other frameworks and systems
- O3.3 Controlled, distributed data access, security, archiving, and storage
- O3.5 Data integrity and redundancy control
- O3.6 Data manipulation and query
- O3.7 Managing relationships between design objects
- O3.8 Data Encapsulation (meaning of encapsulation here seems to be different from some; specifically, it implies import/export)

The two other related goals are:

G4: Tool and framework portability across multiple platforms
- (all)

G7. Extensible to future design complexities and techniques
- O7.4 Scales up with design complexity

Other components of the framework may choose to use the SM's facilities to meet their objectives, but we do not discuss all those possibilities explicitly here.

We will support some of the high-level capabilities mentioned above only at a primitive level, expecting others to contribute at a higher level. These include query, security, data encapsulation (cross-system reference and transfer), archiving, and configuration and design management. We provide leveraged primitives to make higher level implementations possible and easier. The mechanisms we supply enable these higher levels but do not specify policy. For example, the DDM design manager presumably will offer much higher -level checkout capabilities, perhaps using the SM-checkout primitive. Similarly, we will not define a complete high-level associative query language or access mechanism. This is outside the scope of our charter. However, we will supply the low-level primitives required to support standard query mechanisms. In the future we might support adopting some industry-standard query language (e.g., ANSI SQL or its extensions).

For security, the SM should provide primitive services, but the full definition of security capabilities, mechanisms, policy specification, etc., is outside our scope. Similarly outside our domain are: the high-level parts of intersystem transfers, which may occur at the application domain level; archiving, which should involve facilities to specify the scope of the archive, (perhaps application-specific formats and mechanisms); and the full configuration management system, although we will supply the primitives to support versioning at our object level.

The following table lists the detailed requirements for the DDM from reference[1]. We indicate which requirements apply to us and which apply to other framework components or groups. (Note that this document was written in terms of files, while our interface is in terms of objects.)VC indicates the Version/Configuration working groups of DDM; Sec indicates the Security working group of DDM; Sys Env indicates the System Environment TSC. [Although this table references V1.2[1], V1.5[3] has not substantially changed these,

except for, unfortunately, dropping the numbering. These will substantially change when DDM provides new input. At that point, this table will be updated appropriately.]

	SM	Partly SM	VC	Security	Sys Env	Other
8.3.1		9-12			7,8	1-3, 6 Unrealistic
8.3.2	4-5					1-3 Unclear
8.3.3	7		1-6			
8.3.4				1-8, 10		9 Inapplicable
8.3.5		7, 15, 22-24	1-5, 8-14, 16-19, 21, 25-26			6, 20, 27-28 Unclear
8.3.6	3			1-2		

Although the mapping from application-visible requirements to specific requirements and capabilities of the SM can be a confusing, many-to-many connection, we offer the following two sections to define the detailed requirements, in the sense of reference[1], to be considered in the definition of the SMI.

4. SM Design Considerations

Certain SM characteristics and capabilities are critically important to its success, and might be adversely affected or even precluded by an ill-designed interface. We therefore have discussed, among ourselves and with the DDM and DR TSCs, requirements on the SM to ensure that we are not precluding critical capabilities in specifying the SMI.

For example, high performance (speed of access, throughput, etc.) is a critical requirement for the success of the SM. The SMI itself may not specifically accomplish this; however, it must be designed in a way that does not prevent an implementor from achieving such performance goals.

This is a list of some key SM design considerations, which the SMI should not preclude (not ordered):

C1. High performance (read and write)
C2. Large data size (object size, and number of objects)
C3. Checkin/out capability (sometimes called long transactions)
C4. Support for multiple processes sharing data
C5. Sharing of data between versions or perhaps views
C6. High-security implementations

C7. High reliability and robust implementations
C8. Performance optimizations (e.g., to take advantage of locality)
C9. Data integrity maintenance
C10. Recovery
C11. Transaction model
C12. Internationalization support

5. SMI Guiding Principles

We will use the following guiding principles during the development of the SMI. We should judge the degree of our success by these principles. (Not ordered.)

G1. Simplicity. The less complex the design, the better.
G2. Minimal restriction on SM implementations: The SMI should accomplish its goals with the least possible restriction on the implementor of the SM, and so encourage advancement of the state of the art, specialization for different needs, etc., while still achieving interchangeability. In particular, the architecture of the SM should not be specified.
G3. Incremental development of SMI specification: Produce and publish the specification of the non-controversial parts first, in order to achieve progress, even for a subset of the desired goal; then extend the specification to achieve more of the goals.
G4. Compliance levels: publish the specification according to levels of compliance as defined by CFI. We expect at least two such levels.
G5. Interchangeability: Allow SM users to switch to a different SM with the least impact. Switching to a new SM might require, at most, a recompile.
G6. Functionality: Allow support of functionality required by frameworks and applications.
G7. Flexibility: Don't constrain the application's use of the system; e.g., allow it to map objects into its logical entities as it wishes.

The following are highly desirable and could differentiate the levels of compliance (see G4).

G8. Extensibility: Allow extensions to data model and use of the SM; e.g., adding new object types, and dynamic (runtime) type creation/modification.
G9. Multi-process support.
G10. Distribution: Allow support for distributed computing environments.
G11. Heterogeneity: Allow support for heterogeneous computing environments.
G12. Upgradability: Allow support for upgrading designs; e.g., evolution of schemas.

We note that the SM must have access to the type definitions in order to support heterogeneity, upgradability, and the ability to transfer data to different frameworks and archive it. A mechanism for specifying type definitions must be provided within the SMI, in coordination with the DDM Types and Schema group. We hope this mechanism will leverage off existing languages and standards, rather than inventing new ones.

We also note that support of inter-framework transfer mechanisms requires that the SMI support access to all objects. A dump facility, for example, must be able to ask for all objects, regardless of semantics.

6. Functionality Areas

Here we discuss general concepts related to functionality and summarize the areas to be supported. (For complete specification of the functionality see CFI Storage Management High-Level Functional Specification, Browning, ed. 1989).

Unique access to objects should be provided and should include two methods. The first we term OID, or object identifier, and is fully persistent. It references the object uniquely over time and space within some scoping context. The second we term Handle, and is required to be valid only in the context of a single process or group of cooperating processes. This latter is to allow the SM to optimize the performance of dereferencing. Although some implementations might equate handle to virtual memory pointer, the SMI (and the SM user) should not assume this, but rather provide a protected interface (e.g., via macros).

A mechanism to provide scoping of the above OIDs and Handles should be provided.

An efficient relationship mechanism should be provided, to allow one object to reference another object directly, without searching.

Two styles of access to objects should be supported. One is a copy-out approach, in which the application requests the object, and it is copied out of the SM's purview into the application's private memory for safety, and with no more control over the object by the SM. This interface is termed read and write. The other provides a direct reference Handle to the object, to avoid the overhead of the copy, and to allow the SM to retain control for sharing, recovery, etc. This interface is termed open and close. The reason for supporting both is simply that both are needed in different situations. Further, once the latter (open/close) interface is provided, the former (read/write) can be accomplished by a simple layer on top. In that way users of the SM can choose their own trade-offs and policies.

A container concept will be supported. It should include the concept of logical grouping and allow multiple levels of such grouping, but require that an object live in a single container. The intent is to provide a domain-independent mechanism to support application-specific concepts such as a cell, or a schematic page.

Versioning and checkin/out are at the primitive level only. These provide the mechanisms to enable DDM to provide design management functionality. For example, the DDM level may well have a much higher level checkout that deals with the appropriate issues such as configurations, releases, workspaces, namespaces, etc. Similarly, although the SMI supplies the basic capability to create an object (a new "version") given another (and some SM implementations might optimize this), the DDM is expected to deal with the full model and geneology of versions, branches or alternatives, naming, and policy facilities.

File and tool-objects are primitives to enable the DDM to support encapsulation of these entities. For example, the DDM might use this mechanism in support of opening files, maintaining versions, access controls, etc. There is no intent to define (or encapsulate) all the interfaces to the file system; this is presumed to be in the province of the Systems Environment TSC.

The following specific areas of functionality should be addressed:

- Create, destroy, open, close
- read/write
- checkin, checkout (persistent transaction and locking primitives)
- copy, move

- validate
- iterate (traverse a sequence of objects)
- versioning (primitives only)
- transactions
- security (grant, revoke)
- associate, disassociate (relationships, cf. OCT attach)
- type definition
- containment
- (un-, re-)name
- checkpoint
- file objects
- tool objects
- print (dump/load)
- transfer/archive primitives
- miscellaneous, including directives and hints for clustering, etc.

7. SM Documents

The SM Working Group will accomplish its work by publishing the following documents:

A. SM Goals and Requirements (this document)
B. SM Functional Specification (see References, below)
C. SM Interface Specification (to be written)

8. References

1. *CAD Framework Users, Goals, Objectives, and Requirements, Version 1.2*, edited by Paul Painter, Nov. 6, 1989.

2. *CFI Storage Management High-Level Functional Specification - Issue 1.0*, edited by Jason Browning, September 29, 1989.

3. *CAD Framework Users, Goals, Objectives, and Requirements, Version 1.5*, edited by Paul Painter, Nov. 6, 1989.

9. History of this Document

Version 0.1 — Original draft. This document is essentially an extraction from notes and minutes of meetings of the SM Working Group in May, August, and November, 1989, and some email interchanges in between.

Version 0.3 — After review by SM Working Group in Jan., 1990, and approved for distribution to and review by DDM, DR, Arch TSCs and rest of CFI.

Version 0.4 — After initial review by the DDM TSC, 2/28/90. Changes better explain the use of SM by DDM.

Object-Oriented Data Modeling in Rule-Based Software Development Environments

Position Paper

Naser S. Barghouti* and Michael H. Sokolsky†
Department of Computer Science, Columbia University
New York, NY 10027
naser@cs.columbia.edu, (212) 854-8182
sokolsky@cs.columbia.edu, (212) 854-8348

March 29, 1990

1 Introduction

The primary thesis of this position paper is that in order to identify the aspects of object-oriented databases (OODBs) that can lead to standards, more light needs to be shed on the requirements of the applications that use OODBs. The motivation behind developing OODBs was to support advanced applications for which traditional databases are not sufficient. Thus, we have to go back to the needs of these applications and make sure that any candidate for standards meets these needs.

Software development environments (SDEs) generate and manipulate large amounts of data in the form of source code, object code, documentation, test suites, etc. Traditionally, users of such systems managed the data they generate either manually or with the help of special-purpose tools. For example, developers working on a large-scale software project use system configuration management (SCM) tools such as Make and RCS to manage the configurations and versions of the project they are developing. Releases of the finished project are stored in different directories manually. The only common interface between all these tools is the file system, which stores project parts in text or binary files regardless of their internal structures. This significantly limits the ability to manipulate these objects in desirable ways, causes inefficiencies as far as storage of collections of objects is concerned, and leaves data, stored as a collection of related files, susceptible to corruption due to incompatible concurrent access.

*Barghouti is supported in part by the Center for Telecommunications Research.
†Sokolsky is supported in part by the Center for Advanced Technology.

More recently, researchers have attempted to utilize database technology to uniformly manage all the objects belonging to a project. SDEs, for example, need to store the objects they manipulate (design documents, circuit layouts, programs, *etc.*) in a database and have it managed by a DBMS for several reasons: data integration, application orientation, data integrity, convenient access, and data independence.

We have investigated the use of OODBs in a rule-based software development environment kernel called MARVEL [KBS90, KFP88], and have characterized key requirements that the OODB must provide in order to support software development as well as some requirements that are germane to rule-based systems. In this paper we present the application domain and the MARVEL system, and enumerate its data management requirements that we feel should be provided as a standard in OODBs.

2 The MARVEL System

MARVEL is a rule-based development environment kernel that stores software artifacts in a project database, and defines each software development activity that manipulates these artifacts as a rule. Forward and backward chaining on the rules is applied to automate some of chores that developers would have otherwise done manually, to ensures consistency in the project database, and to monitor and/or enforce a particular model of development. The long-term goal of the MARVEL project is to develop a kernel for generating multi-user development environments that use knowledge about the software development process of large-scale projects to support the needs of multiple developers cooperating on these projects.

The kernel is a controlled automation engine that uses a rule-based process model specification and an object-oriented data model specification. These specifications are written in a language called the MARVEL Strategy Language (MSL). MSL specifications are divided into organizational modules called *strategies*. We envision that libraries of MSL strategies will be built, maintained and shared by project administrators. The MARVEL kernel has facilities to *load* and *merge* strategies to produce a target MARVEL environment that understands the data model and process model of the project.

The data model is specified in terms of classes, each of which consists of a set of typed attributes that can be inherited from multiple superclasses. Attribute types include simple types, files, sets and directed links. Set attributes contain instances of other classes as their values, thus implementing *composite objects*, and giving the MARVEL object management system (OMS) a hierarchical traversal capability. Links can be generic, or point to specific attribute types or classes, thus giving the MARVEL OMS arbitrary graph traversal capability. Existing software systems can be immigrated into MARVEL using the Marvelizer tool [Sok89]. The MARVEL OMS supports creation and deletion of objects according to the data model.

The process model defines rules that specify the behavior of the tailored MARVEL environment in terms of what commands are available and what kind of automation is provided. MARVEL supports a model of automation called *opportunistic processing*, which employs backward and forward chaining to automatically initiate activities. The set of rules that are loaded into a MARVEL environment form a network of possible forward and backward chains. MARVEL rules are more complicated then their expert systems ancestors; each rule contains a *precondition* that must be true for the rule to fire, an activity, which is a general

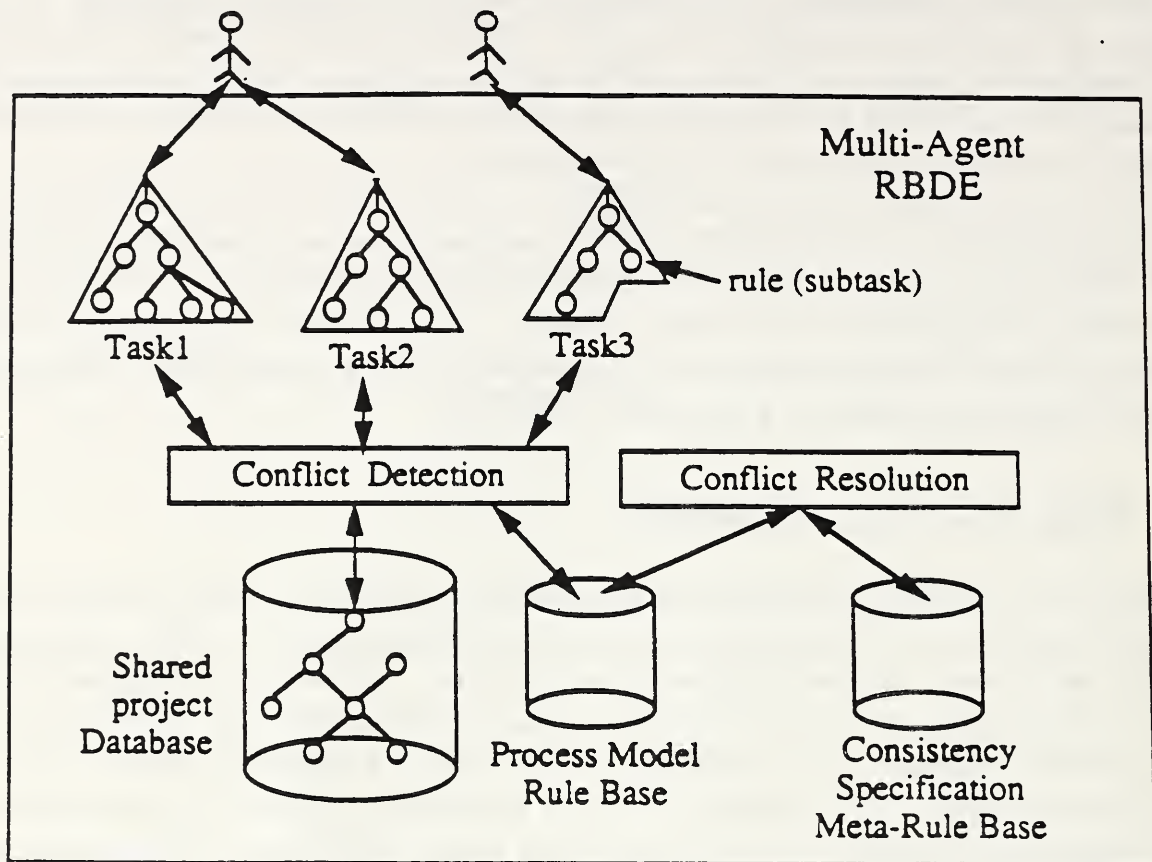


Figure 1: The Multi-Agent problem in MARVEL

mechanism to execute arbitrary, existing tools, and multiple *postconditions* that assert the results of the tool into the MARVEL objectbase.

When multiple developers cooperate on a project within MARVEL, they share a common database that contains all the objects of the project. These developers start concurrent sessions in order to complete their specific tasks. During their sessions, the developers concurrently request operations that access objects in the shared project database. These concurrent operations might violate the consistency of the objects they access if they concurrently change either the same attribute or dependent attributes of the same object in conflicting ways.

Since most operations correspond to rules and since chaining might lead to firing other rules that perform more conflicting operations on the database, more inconsistencies might be introduced in the database. More generally, the overall behavior of cooperating developers in MARVEL can be modeled as multiple sets of rules, where multiple rules from each set are fired concurrently to perform operations on the shared project database. This situation is depicted in figure 1.

In the rest of this paper, we investigate the data management requirements of MARVEL-like software development environments and what facilities are needed to make OODBs meets these requirements. We feel that making these facilities standard features of OODB would would enable OODBs to support the data management needs of advanced applications such

as software development environments.

3 Data Modeling Requirements

MARVEL, like other SDEs, must understand how a project's data is organized in order to provide assistance. In the software development domain, different projects might impose different organization on the data, which in this case consists of the project's components. Thus, MARVEL needs to acquire knowledge about the structure of data in the project under development as well as how this data is accessed. Although OODBs typically support an object-oriented data model, most of them do not fully exploit the features of object-oriented programming. Specifically, they only utilize the structural aspects of object-oritented modeling since they support only aggregation hierarchies (i.e., objects that have the same attributes are grouped together in one class), generalization and specialization hierarchies (i.e., inheritance), and the unique identification of objects.

Existing OODBs do not support the modeling of the behaviorial apects of objects such as the operations that can be perfomed on each object (called *methods* in object oriented languages). Instead the behavioral aspects of the system are modeled in the application that treats objects as passive pieces of data. Thus, rather than providing a uniform and integrated model of data and the operations that can be performed on it, existing OODBs separate the two issues, foregoing the expressive power that can be gained by extending object-oriented features such as multiple inheritance, overriding, and late binding of objects to the development activities.

The project database in MARVEL is defined by a *structurally* object-oriented data model, in the sense of an object data model that defines the organization and structure of the object hierarchy that constitutes the object-oriented database. Our primary innovation in MARVEL is to treat the rules themselves as the "methods" of these objects, and employ *behavioral* concepts from object-oriented programming to integrate the rule system with the shared project database.

In particular, we define the shared project database using an object-oriented data model and further define the methods of these objects via rules, where the rules specify the objects manipulated by each development activity, the condition that must be satisfied to initiate the activity, the tools or other facilities that are employed in carrying out the activity, and the effects of completing the activity with respect to the status of the software project. Developers are assisted by MARVEL's forward and backward chaining on the rules, to automate certain activities, ensure consistency in the project database, and/or monitor developer actions to determine conformance to the designated process and detect divergence from this process.

These behavioral aspects of the data model are typically defined by methods in standard object oriented languages. Methods prescribe the operations that can be perfomed on instances of each class. Each method applies only to instances of the class in which it was defined, and they are invoked by sending a message to an object requesting its invocation. Multi-methods are methods that apply to instances of more than one class. Thus, while attributes describe the structure of objects, methods describe how they behave when they are sent messages from the human developer or other objects. Methods, like attributes, can

be inherited from a class to its subclasses.

MARVEL unifies the behavioral aspects of data modeling with the process model by defining the methods that operate on objects via rules. Each rule has a set of typed formal parameters that are instantiated with objects of the same type (i.e., class) when the rule is invoked. The rule's condition is a logical clause that is essentialy a read-only query on the values of the attributes of the objects that are passed as parameters, while the effects change the values of the objects' attributes. Thus, each rule is equivalent to a multi-method that applies to the classes that are the types of the formal parameters, and, like methods, rules can be inherited. Thus, if a rule has a formal paramter of a certain class C, the objects that are passed as actual parameters can be instances of either C of any of its subclasses.

Based on our experience in MARVEL, we feel that OODBs must provide for both structural and behavioral aspects of data modeling in order to realistically meet the data management needs of applications similar to SDEs.

4 Consistency Maintenance Requirements

The consistency problem has been addressed in traditional database management domains such as banking. In these domains, there is a lack of knowledge about the application-specific semantics of database operations, and a need to design general mechanisms that cut across many potential applications. Thus, the best a database management system (DBMS) can do is to abstract all operations on a database to read and write operations. All computations are then programmed into *transactions* that consist of a sequence of read and write operations. Each transaction, if executed atomically (i.e., either all of its operations are performed in order or none are), transforms the database from one consistent state to another. When multiple transactions run concurrently, the DBMS can guarantee that the database is transformed to a consistent state with respect to reads and writes by allowing only serializable executions of the concurrent transactions [BHG87].

Several existing OODBs use serializability to synchronize concurrent transactions by isolating them, preventing the sharin of data and/or knowledge with other concurrent transactions. Isolation guarantees serializability, and thus strict maintenance of consistency, since it makes each agent's work appear as an atomic transaction. Unfortunately, isolation between concurrent transactions in the software development domain unnecessarily obstructs cooperation [Gre88].

Our conjecture is that an OODB can use knowledge about the structural organization of the project's data, the meaning of data consistency for this project, and the semantics of operations performed by agents on the database in order to provide an extended transaction model. These pieces of information are different for different projects or different phases of the project. For example, the consistency specification of a small project with a couple of developers who are familiar with all the components of the project might permit those developers to access the same object at the same time. The consistency specification of a large project with hundreds of developers, however, might require strict isolation between different groups of developers, and might not allow interaction except through a strictly-defined interface.

Given the flexible consistency maintenance requirements of SDEs, we feel that OODBs

should provide an extended transaction mechanism that supports the following:

1. Long transactions: Long-lived operations on objects in design environments (such as compiling and printing) imply that the transactions, in which these operations may be embedded, are also long-lived. Long transactions need different support than short transactions. In particular, blocking a transaction until another commits is rarely acceptable for long transactions. It is worthwhile noting that the problem of long transactions has also been addressed in traditional data processing applications (bank audit transactions, for example).

2. Composite objects: The complexity of the structure and the size of objects strongly suggest the appropriateness of concurrency control mechanisms that combine multiversion and multiple granularity mechanisms. For example, objects in a software project might be organized in a nested object system (projects consisting of modules that contain procedures), where individual objects are accessed hierarchically.

3. User control: In order to support database operations that are nondeterministic and interactive in nature, the concurrency control mechanism should provide the user with the ability to start a transaction, interactively execute operations within it, dynamically restructure it, and commit or abort it at any time. The nondeterministic nature of transactions implies that the concurrency control mechanism will not be able to determine whether or not the execution of a transaction will violate database consistency, except by actually executing it and validating its results against the changed database. This might lead to situations in which the user might have invested many hours running a transaction, only to find out later when he wants to commit it that some of the operations he performed within the transaction have violated some consistency constraints; he would definitely oppose the deletion of all his work (by rolling back the transaction) in order to prevent the violation of consistency. He might, however, be able to reverse the effects of some operations in order to regain consistency. Thus, what is needed is the provision of more user control over transactions.

4. Synergistic cooperation: Cooperation among programmers to develop versions of project components has significant implications on concurrency control. In CAD/CAM systems and SDEs, several users share knowledge collectively and through this knowledge, they are able to continue their work. Furthermore, the work of two or more users working on shared objects may not be serializable. They may pass the shared objects back and forth in a way that is not equivalent to doing it serially. Also, two users might be modifying two components of the same complex object concurrently with the intent of integrating these components to create a new version of the complex object, and thus they might need to look at each others' work to make sure that they are not modifying the two components in ways that would make integration difficult. To insist on serializable concurrency control in desig environments might thus decrease concurrency or actually disallow desirable forms of cooperation among developers.

5 Conclusions

Object-oriented databases were introduced because of the shortcomings Software development environments are an example of advanced applications that benefit from using an underlying OODB. We presented MARVEL, a rule-based SDE the uses an underlying OODB to store the software artifacts of projects under development. We presented some of the data management and data consistency requirements of SDEs that we have characterized based on our experience with MARVEL. We believe that these requirements are braod enough to be applicable to many advanced database applications, and thus they have to be considered in any effort leading to the formal standardization of OODB.

References

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Readin MA, 1987.

[Gre88] I. Greif, editor. *Computer-Supported Cooperative Work: A Book of Readings*. Morgan Kaufman, San Mateo CA, 1988.

[KBS90] G. E. Kaiser, N. S. Baghouti, and M. H. Sokolsky. Preliminary experience with process modeling in the marvel software development environment kernel. In *23rd Annual Hawaii International Conference on System Sciences*, Kona HI, January 1990.

[KFP88] G. E. Kaiser, P. H. Feiler, and S. S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[Sok89] M. H. Sokolsky. Data migration in an object-oriented software development environment. Master's thesis, Columbia University Department of Computer Science, April 1989, Technical Report CUCS-424-89.

An Entity-oriented Data Model - MIX

Tzy-Hey Chang
Applied Intelligent System Group
Digital Equipment Corporation
chang@aisg.enet.dec.com

March 9, 1990

1 Introduction

The rapid evolution of computer technology has widened the spectrum of end-users, overfilling the capacity of data processing departments to satisfy their demands [7]. One of the central issues in database research is how to provide an environment for the end-user to manipulate a database directly, instead of depending on professionals. The semantic data model (SDM) [12], the relational data model (RDM) [8,32,9] and the universal relation model (URM) [32,18] all represent attempts at solving this problem.

The semantic data model extends the power of a standard programming language (such as PASCAL) to be a database language (such as PASCAL/R [25]) with semantic constructs [23,14,28,15]. However, because the nondeclarative flavor of the conventional programming language, the user usually has to write programs to navigate around the database contents in order to ask a query. By contrast, the set manipulation operators in the RDM relieve the user from navigating through each relation tuple by tuple, but the user still has to explicitly connect relations together in order to construct a query. The URM further relieves the user from this higher-level navigation; a user query only needs to mention the attribute names of interest, not the relation names. Thus, the URM provides a more user-friendly interface.

However, the URM requires the database designer to enforce some rigid assumptions [21] which need to check on database scheme and database globally, the characteristic of those assumptions causes difficulties in the database designs and database maintenance. In this paper, we propose a new data model, called *MIX Semantic Data Model*, which keeps the URM interface, while eliminates the limitations (assumptions) of the URM. The proposed data model incorporates available semantic concepts from the SDM such as entities, ISA hierarchies, aggregation hierarchies, and characteristic property hierarchies, shared property hierarchies into the URM to model a database in a top-down modular way. The modularized entity-oriented data model allows us to deal with the ambiguity of query interpretation locally rather than globally, hierarchically rather than strictly enforcing assumptions. The model is able to provide a URM-style's query language for database retrievals and updates [3]; in contrast, previous URM approaches can only deal with database retrievals. In addition, the graphic representation of these semantic constructs provides a framework for implementing an integrated graphic user interface for database design, manipulation and maintenance. Because of the page limitation, the discussion of query language, update semantics, the modularization of data model are discussed in [3].

The paper is organized as follows. Section 2 describes the semantic constructs , module constructions, and the local unique name restriction, the attribute correlation assumption in the MIX. Section 3 describes query interpretation. Section 4 provides the summary and indicates future directions.

2 The MIX Entity-Oriented Data Model

In this section, we shall explain our semantic data model, the MIX. Although there are many semantic data models in the literatures [29,4,34,31,17,5,23,14,28,15], ours is based on RM/T [5], IFO [1] and ERM [4]. What significantly distinguishes our approach to their is that the main goal of our data model is to provide a URM interface in which user queries can be interpreted unambiguously.

There are four kinds of semantic constructs in the MIX: *ISA hierarchies*, *characteristic property hierarchies*, *aggregation hierarchies*, and *shared property hierarchies* which will be discussed in Sections 2.1 - 2.4. Through these semantic constructs, the interaction among entities in the world can be explicitly represented in graphic notations. These semantic constructs enable us to improve the readability of the database schema of the previous approaches [11,20,21,22,27,26,24,33,19,16], for instance, the specifications of associations and objects in [19] are hard to read as things get complex. The graphic notations also facilitate us to implement an integrated graphic user interface for the entire database system life cycle.

2.1 Entities, Roles, ISA Hierarchies

In our model, the notion of entities is the basic building block to model the real world. An *entity* is something can be uniquely identified in the world we want to model, such as employee, secretary or customer_order. Entities are modelled by one or more *objects* in the database, where each object belongs to a particular *class*. For example, a particular *Employee-entity* might be modelled by objects in classes *Employee, Manager, Stockholder* and so on. Each class defines a set of *attributes*. Each attribute maps an object in that class to a value. In addition to the property attributes of the entities, each class has a surrogate attribute. The name of the surrogate attribute is the class name concatenated with the special character #. When a new object is inserted into the relation for the class, a corresponding surrogate value is generated by the system. Thus surrogate attribute can be used as the primary key of the class.

Although an entity may be modelled by several objects, each entity has one object which is considered to be its *representative*. For example, the representative for the above entity is the object in *Employee*. The class *Employee* is an example of a *representative class*. Each object in a *representative class* corresponds to a unique entity, and inserting (or deleting) objects in this class models the insertion (or deletion) of an entity in the database. Objects which are not representative can be thought of as *roles*, and their classes are called *role classes*. Each role object is associated with a single object in the representative class, and denotes additional (non-essential) information about the entity. An entity can have several role objects, and this set of objects can change dynamically without affecting the existence

of the entity. For example, some employees are stockholders, some are managers, and some may be both. Furthermore, the deletion of an object from *stockholder* class is not modelling the deletion of an entity, but a change in roles of an existing one. However, deleting a representative object causes all of its associated role objects to be deleted as well. Given a representative class, each of its objects may play a different set of roles. For example, some employees are stockholders, some are managers, and some may be both.

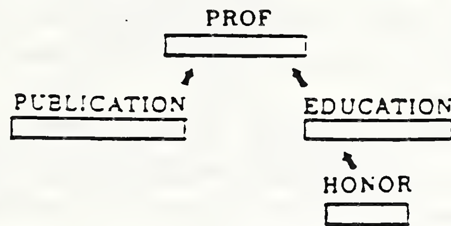
A representative class and all its associated role classes form an *ISA hierarchy* which is a directed graph with classes as nodes and existence dependency edges ⇢ as directed edges. The existence dependency edge points from subclass to its direct superclass to indicate that the existence of objects in the subclasses depends on the existence of objects in its direct superclass, but not vice versa. For clarity, we will use different arrows to denote existence dependency constraints in different semantic constructs of the MIX: ➡ for characteristic property hierarchies, ⇏ and ⇒ for shared property hierarchies, ⇢ for ISA hierarchies, and → for aggregation hierarchies. A class is represented by a rectangular-shaped box with the class name attached to the left-corner of the box.

Duplicate attribute names cause ambiguous query interpretation, because they can be mapped into any classes with those attribute names. Therefore, filtering out common attributes is a necessary step in the process of generalization/specialization. Common attributes among role classes are moved up the hierarchy, so that there is never any duplication of attribute names. As long as there is a duplication, we can do more generalization. Because applying generalization/specialization on the same set of entities for modelling an ISA hierarchy, we assume that an entity can only appear in one ISA hierarchy.

ISA hierarchies in our model must be single rooted, acyclic and non-redundant. Multiple rooted ISA hierarchies mean that most-generalized representative class do not form, and duplicate attribute names may exist among classes of ISA hierarchies. We require that there be no cycles in an ISA hierarchy, because cyclic existence dependency constructs among subclasses are meaningless. The requirement of non-redundancy means that there can be no transitive edges in an ISA hierarchy. Relationships can be formed among classes of ISA hierarchies (Section 3.1).

2.2 Characteristic Property Hierarchy

A ***characteristic property hierarchy*** [5] shows the relationships among multivalued attributes and the entities described by them. Single-valued attributes on class A are represented graphically inside its class box directly. For multivalued attributes of class A, we create another class B (characteristic class) whose existence is dependent on the class A. Obviously, the relationships between objects in A and B are 1:n relationships. We may need to further decompose B if B itself contains other nested multivalued property attributes to describe it. The entity decompositions result in a tree structure [5], called characteristic property



PROF(PROF#,ID,PROF_NAME,PROF_AGE).
EDUCATION(EDUCATION#,PROF#,SCHOOL_NAME,DEGREE,GRADUATION_DATE).
HONOR(HONOR#,EDUCATION#,HONORS,HONOR_DATE).
PUBLICATION(PUBLICATION#,PROF#,TITLE,JOURNAL,PUBLICATION_DATE).

Figure 1: The characteristic property hierarchy for professors

hierarchy. The existence dependency constraint for B on A is represented as an existence dependency edge ➡ pointing from B to A, and the class B contains the surrogate attribute of class A to show the relationship between classes A and B, as shown in Figure 1. The *root node* in a characteristic property hierarchy is a *representative class* if there are no existence dependency edges pointing out from it. Other classes in a characteristic property hierarchy are called *characteristic classes.*

Example 1 Suppose the entities PROF have the single-valued property attributes (ID, PROF_NAME, PROF_AGE), multivalued property attributes about professors' education (SCHOOL_NAME, DEGREE, GRADUATION_DATE, HONORS, HONOR_DATE) and multivalued property attributes about professors' publication (TITLE, JOURNAL, PUBLICATION_DATE). The entity decompositions result in a characteristic property hierarchy that is a tree structure as shown in Figure 1, where the scheme for each class is shown in the lower part of the figure. Note that we decompose multivalued property attributes about professors' education into classes EDUCATION and HONOR, because professors might get more than one honor in each school.

2.3 Aggregation Hierarchies

In [5,29], an aggregation may mean the aggregation over properties to form a class or the aggregation over classes to form a higher level aggregative class. Our approach is the same as the latter case. The aggregative classes is represented as a diamond shape inside

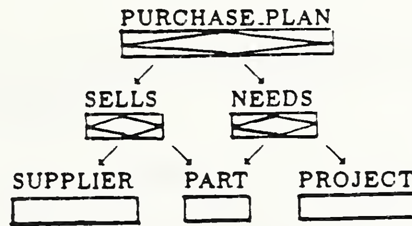
a rectangular shaped box with existence dependency edges → pointing to the underlying component classes. There is more than one surrogate attribute attached to the aggregative class; one is the identifier for the aggregative class; others are the identifiers for the direct underlying component classes.

An aggregation hierarchy is required to be an acyclic graph as illustrated in Figure 2 because cyclic component relationships among different aggregative classes need property non-applicable null values to terminate these cyclic relationships among entities. As we know, the answer of a query interpretation [6] which is able to deal with property unknown and property non-applicable null values may confuse the users. ISA hierarchies discussed in Section 3.1 can be used to construct acyclic relationship graph. The existence dependency edges pointing from an aggregative class A to its component classes B's specify the existence dependency constraints, where the existence of objects in the aggregative class depends on the existence of objects in its component classes. The relationships between tuples in A and B are n:1 relationships. Every aggregative class is a *representative class*.

The abstraction of aggregations results in an acyclic graph that is not necessarily single rooted. Our data modelling for unambiguous query interpretation needs the notion of single rooted aggregation hierarchies that are the subgraphs of the acyclic graph. In our approach, an aggregation hierarchy consists of a *root node*, called *root aggregative class*, (an aggregative class that does not further participate in any relationships), and the direct and indirect component classes of the root node. The root node will play a special role in the query interpretation, as discussed in Section 2.6. A multi-rooted acyclic graph is considered to be several (single-rooted) aggregation hierarchies whose nodes may overlap. For instance, if we remove the class PURCHASE_PLAN from Figure 2, then there are two aggregation hierarchies, and PART is the overlapping node between them.

2.4 Shared Property Hierarchies

In Section 2.2, we saw how characteristic property hierarchies model the relationships among entities and their individual multivalued properties. That semantic construct is used to avoid update anomalies. Here, we introduce a new semantic construct, called a *shared property hierarchy*, for the same purpose. A shared property hierarchy shows the relationships among entities and their common, shared properties. That is, we decompose the shared properties for the entities in a class A as a separate class C, and create a *linking class* B whose tuples are used to connect the tuples between A and C. Class A is the *main representative* class, and C as the *shared property class* (which is also a representative class, see below). The classes A, B and C are defined as a *shared property hierarchy*. The arrow ⇻ pointing from the linking class to the main representative class, and the arrow ⇒ pointing from the linking class to the shared property class denote the existence dependency edges. The example below demonstrates the difference between these two semantic constructs.

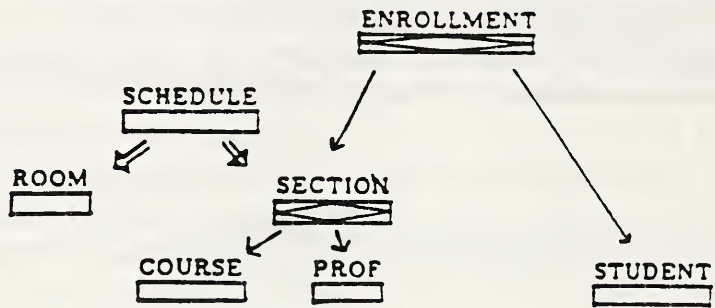


SUPPLIER(SUPPLIER#,SUPPLIER_NAME,ADDR,TEL).
PART(PART#,PART_NAME,DESCRIPTION).
PROJECT(PROJECT#,PROJECT_NAME,DEADLINE).
SELLS(SELLS#,SUPPLIER#,PART#,PRICE).
NEEDS(NEEDS#,PART#,PROJECT#,USAGE_QTY).
PURCHASE_PLAN(PURCHASE_PLAN#,SELLS#,NEEDS#,ORDER_QTY).

Figure 2: The aggregation hierarchy for classes SUPPLIER, PART and PROJECT

Example 2 Suppose a student enrollment database scheme contains the following classes for course-related information. The graph representation is shown in Figure 3 in which the attributes ROOM_NO and SEATS are the common, or shared properties for the entities in SECTION, so they are decomposed from SECTION to be a separate class ROOM. The entity decomposition results in a shared property hierarchy.

If all information about course schedules is represented in a class such as SEC(SEC#, COURSE#,PROF#,SEC_NAME,DAY_HOUR,ROOM_NO,SEATS), there must exist redundant information about SEATS which will cause update anomalies. The semantic construct of a characteristic property hierarchy cannot solve the above update anomalies. For instance, suppose we decompose SEC into SEC(SEC#,COURSE#,PROF#, SEC_NAME,DAY_HOUR) and ROOM(ROOM#,SEC#,ROOM_NO,SEATS). Then, the information about rooms is duplicately recorded for each course section using the same room. The information about rooms is the *shared* property for the entities in the representative class SEC, so an update on a room must reflect on all sections using this room. In contrast, a characteristic class in a characteristic property hierarchy represents the *individual* properties for the entities described by them. Thus, a characteristic property hierarchy is not suitable for our purpose here. We need to decompose SEC into SECTION, SCHEDULE and ROOM as shown in Figure 3, in which the tuples in SCHEDULE are used to connect the tuples between SECTION and ROOM. This example also demonstrates why a linking class is needed in a shared property hierarchy.



COURSE(COURSE#,COURSE NAME,DES).
PROF(PROF#,PROF_NAME,RANK,PROF_ADDR).
STUDENT(STUDENT#,STUDENT NAME,YEAR,STUDENT_ADDR).
SECTION(SECTION#,COURSE#,PROF#,SEC_NAME).
SCHEDULE(SCHEDULE#,SECTION#,ROOM#,DAY_HOUR).
ROOM(ROOM#,ROOM_NO,SEATS).
ENROLLMENT(ENROLLMENT#,SECTION#,STUDENT#,GRADE).

Figure 3: A student enrollment database scheme

2.5 The Difference among Characteristic, Shared Property and Aggregation Hierarchies

The shared property class can independently participate in other relationships or other shared property hierarchies. For example, we may declare DEPARTMENT as the shared property class for the entities in each class COURSE, PROF and STUDENT. In this case, we form three new shared property hierarchies. It implies that the shared property class can exist independently of the main representative class. Therefore, the shared property class is a representative class as well in the MIX. The characteristic class can not overlap or participate in other hierarchies. For query interpretation, the classes in shared property hierarchy are treated as a unit of abstraction for the objects in their main representative class; this is like the classes in a characteristic property hierarchy are treated as a unit of abstraction for the objects in their representative class.

Since an object in a shared property class can exist independently of its root representative object, its update semantics is different from that of an aggregative class [3]. For instance, suppose PURCHASE_PLAN class is deleted from Figure 2 and suppose PART, PROJECT and NEEDS define a shared property hierarchy, where NEEDS is the linking node, PROJECT is shared property node. Then, a deletion request on the new database scheme << SUPPLIER = s1, PART = p1, PROJECT = pj1 >> will only delete an object in SUPPLIER_PART, because we treat the attributes in PROJECT as the shared properties for the objects in PART. In contrast, the same user request applied on Figure 2 will delete an object in SUPPLIER_PART_PROJECT. Therefore, the intended update semantics may cause the database designer to choose different data modellings [3]. This is similar to the way objects are defined over associations in the [19]. However, an object in the [19] does not carry information to guide the update behavior.

The relationships between objects in the representative class and the shared property node of a shared property hierarchy are m:n relationships, whereas the relationships between objects in the representative node and the characteristic class of a characteristic property hierarchy are 1:n relationships. The linking class in a shared property hierarchy is similar to an aggregative class in an aggregation hierarchy. However, each aggregative class in an aggregation hierarchy is treated as a unit of abstraction and can further participate in other relationships, whereas a linking class cannot participate in other relationships.

2.6 Modules

According to the methodology and restriction for data modelling discussed so far, we can represent a database scheme D = (R, C) with classes R and integrity constraints C as a

directed acyclic graph G = (V,E), called a *database schema graph*, which consists of a finite set of vertices V and a finite set of edges E ($E \subseteq V \times V$). If $<v_1,v_2>$ is in E, we say that the edge is directed from vertex v_1 to vertex v_2, or v_2 is adjacent to vertex v_1. For each $v \in V$, there is a corresponding class $r \in R$. For each ordered pair $<v_1,v_2>$ in E, where v_1, v_2 correspond to r_1, r_2 in R, there is a corresponding existence dependency constraint for r_1 on r_2 in C. Thus, each ISA hierarchy, aggregation hierarchy, characteristic property hierarchy or shared property hierarchy is a subgraph of G and the vertices for them may be overlapped. For instance, in Figure 3, the class SECTION is the overlapping node between an aggregation hierarchy and a shared property hierarchy.

Our approach to reduce the intricacy of the interaction among classes in different semantic constructs, as described in the above, is to organized *related abstractions or concepts* into one *module*. As far as module constructions, the reader may refer to [3].

Definition 1 Each representative class has an associated set of classes, which is called its closure. The *closure* of the representative class A, denoted by A^+, is the union of {A} and the closure of the classes in the following set:

1. The set of classes in the same characteristic property hierarchy as that of class A.
2. The set of classes in the same shared property hierarchy as that of class A.
3. The set of classes in the same ISA hierarchy as that of class A.
4. The set of A's component classes if A is an aggregative class.

The set of attributes for the classes in A^+ is denoted by $Att(A^+)$.

Definition 2 The closure of the root aggregative class defines a *module*.

In the MIX, because related concepts are organized into one module, we can assume that the attributes in the user window are from only one module. Based on the concepts of modules, we propose a *local unique role restriction* to replace the *global unique role restriction* in the URM. Note that the global unique role restriction (global attribute renamings) can expand the universe of the attributes significantly, making it difficult for a end-user to remember them. The local unique role restriction only requires that an attribute must play a unique role within a module, but not within the entire database scheme. Thus, it is able to solve the problem caused by the global attribute renaming.

3 Query Interpretation in Our Approach

In the MIX, the interpretation of a query is based on the notion of of a *context* for the user window. Roughly, we need to find a representative class whose abstraction contains the minimum abstraction of attributes in the user window. If a query has more than one minimum abstraction, then it is considered to be ambiguous.

Suppose X is the set of attributes in the user window. Suppose there are two representative classes M and N, such that $X \subseteq Att(M^+)$, $X \subseteq Att(N^+)$ and $Att(M^+) \subseteq Att(N^+)$ are true. $X \subseteq Att(M^+)$ (or $X \subseteq Att(N^+)$) implies that X can be derived from $Att(M^+)$ (or $Att(N^+)$). Also, $Att(M^+) \subseteq Att(N^+)$ implies that N is a higher level abstraction based on M in the module for the user window. For query interpretation, we assume that M^+ is closer to the user's intent than N^+. If there is no representative class E such that $X \subseteq Att(E^+)$ and $Att(E^+) \subseteq Att(M^+)$, then M is called the *context* for the user window. This is similar to the case of natural language, where we interpret the utterance of a person according to what he says, with no extra context is added.

Because of the entity decompositions occurred in characteristic property, shared property, and ISA hierarchies, MIX are able to represent a set of entities with heterogeneous properties without using property non-applicable null values. Thus, different subsets of classes in these three hierarchies can determine different subsets of entities represented by them. For instance, in Figure 1, the relations for user query [PROF_NAME,TITLE] and [PROF_NAME,DEGREE] should represent different subsets of entities. For query evaluation, we map the attributes in the user window to the nodes in the database schema graph, those nodes are called *explicitly bound classes*. The explicitly bound classes for the first query in the above example are PROF and PUBLICATION; the explicitly bound classes for the second query are PROF and EDUCATION. The join of classes in $PROF^+$ - that is, {PROF, PUBLICATION,EDUCATION,HONOR} - will not give a correct query evaluation. Therefore, we need to compute the *minimum connection* among bound classes to distinguish the subsets of entities in these three hierarchies [3]. For instance, the above PROF example is evaluated as follows. [PROF_NAME,TITLE] is evaluated as PROF $\bowtie$ PUBLICATION; [PROF_NAME,DEGREE] as PROF $\bowtie$ EDUCATION.

3.1 Recursive Relationships

The algorithm for complex queries such as recursive relationships is explained briefly below; see [3] for details. A relationship between different objects in the same set of entities is called a *recursive* relationship. In our approach, if a relationship is between disjoint subsets A1 and A2 of a set of entities, we can form an ISA hierarchy between A, A1 and A2 to form a relationship between A1 and A2. Although the approach produces a nice, clear data model, a user query may reintroduce an attribute playing multiple roles, as illustrated in

the example below. In this case, our query interpretation needs modification.

With respect to a unique context M for the user window [X], if M^+ contains more than one node o_i overlapping an ISA hierarchy and the aggregation hierarchy, the meaning of o_i is o_i and its explicitly bound superclasses, but not other classes in the same ISA hierarchy. This semantics conform to the purpose of data modelling that distinguish roles in order to avoid recursive relationships. The approach has one advantage. Because it spans each o_i to its superclasses, the user can always get his desired answer. If explicitly bound role classes can not be appropriately classified according to the path between o_i and its explicitly bound superclasses, then the query interpretation is ambiguous. Focal graph described in [3] is used for disambiguating query interpretation.

4 Summary and Future Directions

Our goal is to make the data in the database more accessible to end-users. The MIX entity-oriented data model enables us to provide a consistent URM-style's query language to allow the users to express a database updates and database retrievals by mentioning attribute names and their values only [3]. We use focal graph to interact with the user in case ambiguity occurred in the query interpretation. Our data model provides a framework for a more user-friendly interface. An integrated graphic user interface for database design, data definition and data manipulation is being implemented. Dynamic SQL, DECwindow widgets and callback routines facilitate the information passing among database manager, window manager, and host language modules. For instance, at the database design phase, DECwindow callback routines pass user's inputs to dynamic SQL to generate SQL commands.

We will provide a mechanism to allow the end-users to express transitive closure relationships in a natural and intuitive way. In [30], end-users have to specify a repetitive command and a temporary relation for the query "to find all of the employees who work for Joe". The retrieval will repetitively execute until there are no changes in the temporary relation. This is not easy for a casual user. We propose to use the notations (↑, ↓) appended to attribute names for expressing transitive closure relationships. This approach is consistent with our data manipulation language in which end-users only have to specify attribute names. For instance, the query [MANAGER_NAME↑(*) | NON_MANAGER_NAME = Tom, GROUP_NO = 2], will return all different levels of bosses of Tom in group 2; the query [MANAGER_NAME↑(2) | NON_MANAGER_NAME = Tom, GROUP_NO = 2] will return the bosses of the bosses of Tom. For the graphic user interface, the end-users can enter the notations (↑ (n), ↓ (n)) to attribute column to express queries on transitive closure relationship, the system will display the hierarchical transitive closure relationships in a navigational widget.

For database access, insertion and deletion are not necessarily reversible [27,13,10]. For providing facilities for end-users to undo update effects, we propose to include a transaction number attribute into each object of the database scheme. When an update command is initiated, a transaction number is generated by the system for those affected tuples. An undo command issued by end-users can correctly reverse update effects in terms of tuples with the same transaction numbers. The undo command is different from the undo in [32,9] that is invoked by the system to roll back a failed transaction. Because transaction numbers are based on time, how to incorporate this facility to the MIX for historical database inference or version control is a future research area.

References

[1] S. Abiteboul and R. Hull. IFO : A foraml semantic database model. *ACM Trans. on Database Systems*, 525–565, December 1987.

[2] C. Beeri and H. F. Korth. Compatible attributes in a universal relation. In *Proc. ACM. Symp. Principles of Database Syst.*, pages 55–62, March 1982.

[3] T. H. Chang. *A Universal Relation Data Model with Semantic Constructs*. PhD thesis, Boston Univ., 1988.

[4] P. P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. on Database Systems*, 1(1), 1976.

[5] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. on Database Systems*, 4(4):397–434, 1979.

[6] E. F. Codd. More Commentary on Missing Information in Relational Databases (Applicable and Inapplicable Information). *SIGMOD RECORD*, (1):42–50, March 1987.

[7] E. F. Codd. Relational Database : A Practical Foundation for Productivity. *CACM*, 25(2), February 1982.

[8] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[9] C. J. Date. *An Introduction to Database Systems*. Addison-Welsey, 3 edition, 1982.

[10] U. Dayal, P. A Bernstein. *On the updatability of relational views*, *Proc. 4th VLDB Conf. IEEE Computer Society*, March, 1978.

[11] R. Fagin, A. O. Mendelzon, and J. D. Ullman. A simplified universal relation assumption and its properties. *ACM Trans. Database Syst.*, 7:343–360, 1982.

[12] R. Hull and R. King. *Semantic Database Modeling : Survey, Applications, and Research Issues.* Technical Report, University of Southern California, Comput. Sci. Tech. Rep. TR-86-201, April 1986.

[13] A. M. Keller. *Updating relational databases through views*. PhD thesis, Stanford Univ., 1985.

[14] R. King and D. McLeod. An Approach to Database Design and Evolution. In M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *Data Abstraction, Databases, and Conceptual Modeling*, 1984.

[15] R. King and D. Mcleod. The event database specification model. In Peter Scheuermann, editor, *Improving database usability and responsiveness*, Academic Press, 1982.

[16] H. F. Korth, G. M. Kuper, A. Feigenbaum, V. Gelder, and J. D Ullman. System/U : a database system based on universal relation assumption. *ACM Trans. on Database Systems*, 9(3):331–347, 1984.

[17] Y. E. Lien, J. E. Shopiro, and S. Tsur. DSIS - A Database System with Interrelational Semantics. In *7th Int. Conf. VLDB*, pages 465–477, 1981.

[18] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[19] D. Maier, D. Rozenshtein, S. Salveter, J. Stein, and D. S. Warren. Toward logical data indenpence : A relational query language without relations. In *ACM SIGMOD Intl. Symp. on Management of Data*, pages 51–60, June 1982.

[20] D. Maier and J. D. Ullman. Maximal objects and the semantics of universal relation databases. *ACM Trans. Database Syst.*, 8(1):1–14, 1983.

[21] D. Maier, J. D. Ullman, and M. Y. Vardi. On the foundations of the universal relation model. *ACM Trans on Database Systems*, 9(2):283–308, 1984.

[22] D. Maier and D. S. Warren. Specifying connections for a universal relation scheme database. In *Proc. ACM SIGMOD Intl. Symp. on Management of Data*, pages 1–7, June 1982.

[23] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM Trans. Database Systems*, (2):185–207, June 1980.

[24] Y. Sagiv. Can we use the universal instance assumption without using nulls? In *Proc. ACM SIGMOD Intl. Symp. on Management of Data*, pages 108–120, May 1981.

[25] J. W. Schmidt. Some high level language constructs for data of type relation. In *ACM TODS, 2, 3*, pages 247–261, September 1977.

[26] E. Sciore. Inclusion dependencies and the universal instance. In *ACM Symp. on Principles of Database Systems*, pages 48–57, 1983.

[27] E. Sciore. *The universal instance and database design.* PhD thesis, Princeton Univ., 1980.

[28] D. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. on Database Systems*, 6(1):140–173, 1981.

[29] J. M. Smith and D. C. P. Smith. Database abstractions : Aggregation and generalization. *ACM Trans. on Database Systems*, 2(2):105–133, 1977.

[30] M Stonebraker, R. Johnson, and S. Rosenberg. A rule system for a relational data base management system. In Peter Scheuermann, editor, *Improving database usability and responsiveness*, Academic Press, 1982.

[31] S. Tsur and C. Zaniolo. An implementation of GEM - supporting a semantic data model on a relational back-end. In *Proc. ACM SIGMOD Intl. Conf. on the Management of Data*, pages 286–295, 1984.

[32] J. D. Ullman. *Database Systems*. Computer Science Press, 2 edition, 1982.

[33] J. D. Ullman. The U. R. strikes back. In *Proc. ACM Symp. on Principles of Database Systems*, pages 10–22, March 1982.

[34] C. Zaniolo. The database language GEM. In *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, pages 207–217, 1983.

[35] M. M. Zloof. Query-by-example : A data base language. *IBM Syst. J.*, 16(4):324–343, 1977.

Object Data Model = Object-Oriented + Semantic Models

Qing Li

Department of Computer Science
Australian National University
Canberra, ACT 2601

Abstract

There has been a lot of confusion about the definition of "object" data model in the database community. People use the term "object database" to refer to different characteristics and mechanisms which are typically embodied in so-called "semantic" databases and "object-oriented" ones. While these two classes of data models stem from different roots and applications, they nevertheless resemble each other in many aspects, and complement each other in many ways. This paper summarizes from these two paradigms the features common to both, the features influenced/adopted by each other, and and the features which are present in one paradigm but missing from the other. After a brief analysis of these features, we conclude the paper with some discussions on the desirability of combining these two sorts of models together, and propose the major theme of the paper: a complete object data model should take the union of the facilities offered by these two classes of models.

1 Introduction

Over the last decade, we've been seeing a dramatic proliferation of systems and models which are claimed to be (or labeled with) "object-oriented". As pointed out in [10], the term "object-oriented" has been serving as an effective yet false advertising for selling ideas, models, and systems in database community. On one hand, this reflects the common interests and desires in developing more advanced database technology for non-traditional applications. On the other hand, it also reveals the common confusion about the concept of object, and the definition of an object data model.

The main reason for such a chaos is, of course, due to the lack of a common data model, and the lack of formal fundations [2]. This is natural since object paradigm is still a relatively young and developing technology. Indeed, there are still a lot of experimental work underway. Another reason is probably due to the historical development of object databases. After people realized the shortcomings of relational databases for more advanced applications, significant research efforts have been devoted to

either extending the relational model to capture more meanings [4], or to develop new data models which are semantically more expressive and powerful [3, 6, 7]. The latter is collectively referred as "semantic" database models, or "structurally object-oriented" models [5]. Orthogonal to that is the more recent development of so-called "object-oriented" databases, which have been inspired from object-oriented programming languages [11, 1, 8]. This type of databases are also sometimes referred as "behaviorally object-oriented" [5], indicating the emphasis of such databases on behavior modeling. But quite often, people don't call their models and systems to be "structurally" or "behavirally" object-oriented, but rather they just call them (for simplicity and/or for selling purpose) "object-oriented", which is where the confusion really stepped in.

Being aware of such confusion, some initial efforts have been attempted on this issue (e.g., [5, 2, 10]). Such efforts include those trying to identify/claim what features pertain to object-oriented databases, and/or to clarify certain differences between these two types of data models (with strong bias on "object-oriented" ones). As a consequence, the results seem to either underrate the importance of semantic models to object-oriented ones, or simply ignore the influences from semantics models. In this paper, we take a different attitude towards this issue. In our viewpoint, semantic models and object-oriented ones are not only closely related to each other, but also inseparable from each other when the concept of object is to be defined. Admittedly, putting them all under the name "object-oriented" can be misleading. However, simply split semantic models from object-oriented models is also inappropriate, nor desirable. Indeed, object-oriented databases have benefitted from semantic database modeling in many aspects, with semantic models also being influenced by object-oriented ones in some others. Putting the political factors aside, the important thing for us to consider is, of course, whether it is desirable to put them into the same family. Will it be a loss for object-oriented databases if semantic models are excluded from the family? Can these two types of models be integrated together naturally and effectively? This paper attempts to address on these questions, by comparing and contrasting features from both classes of models. First, we list the common/similar features to both classes; next, we examine the features which are influenced/adopted from each other class; we then consider the features specific to each class, concentrating on the complementary and/or conflict aspects (if any) to each other. We conclude the paper with discussions on the desirability of integrating these two sorts of models together, and propose the major theme of the paper—a complete object data model should encompass both semantic and object-oriented model facilities.

2 Semantic and Object-Oriented Models: A Comparative View

There have been a large number of semantic data models introduced since the mid-70's, and object-oriented models introduced since the mid-80's (see the survey papers or collection of readings in [7, 12, 9, 13]). Various models and systems emphasize on various aspects and features. For comparison

purpose, we shall focus on essential (or mandatory) features [7, 2]) for each type of models, and use the short-hand SDB as a generic representative for semantic databases, and short-hand OODB for object-oriented databases.

2.1 A Brief History of SDB and OODB

To compare and contrast SDB and OODB, we first consider briefly on their development histories and motivations. SDB was originally introduced primarily as schema design tools for databases, the emphasis of which was to accurately model data relationships that arise frequently in typical database applications. It has been influenced by the work on knowledge representation in AI (e.g., the semantic networks and frames). On the other hand, OODB was inspired by advances in programming languages—the object-oriented ones, stemming from research on abstract data types (ADTs). Consequently, SDB has been emphasizing on the representation of data, and OODB is geared towards the manipulation of data. Despite the fact that they grew out of different "roots", and had different emphases, they have now become closely related to each other, and are virtually inseparable from each other, as the subsequent discussion shows.

2.2 Common/Similar Features

What seems to be interesting and important to observe in comparing SDB and OODB is that although they are fundamentally different, they resemble each other in many ways. This indicates some common sought-after features and capabilites desired by people from different applications and with different purposes. The following is a list of features which are believed to be common (or similar) to both SDB and OODB.

2.2.1 Object Identity

The concept of object identity has long existed in programming languages, but was not introduced into databases until the mid-70's when the RM/T model and other early semantic data models were developed. Developed from combining programming languages with databases, object-oriented databases have also inherited this feature into their systems.

2.2.2 Types/Entities/Classes

Modeling on semantic databases have introduced several data abstracions, which include the *generalization*, *aggregation*, *classification*, and *association*. Classification gives us the concepts of "class" (or "entity"). On the other hand, type mechasims in programming languages have provided similar concepts of "type" (in some languages also called "class"). While these concepts vary in some aspects, they all capture the major semantics of "is-instance-of" or "is-member-of".

2.2.3 Relationships

Both SDB and OODB have facilities for modeling general inter-object relationships. In SDB this is captured by the association mechasim, and in OODB, this is via specification of object properties.

2.3 Features Adopted/Influenced by SDB and OODB

Besides all the coincidental common/similar features introduced by SDB and OODB separately, there are also features which are influenced from each other. Here, we briefly examine such interactions between these two paradigms.

2.3.1 Aggregation

As we mentioned above, data modeling on SDB has introduced several important and useful data abstractions. Aggregation is introduced to model the object which has several parts (ie., the "is-part-of" relationship). This was later adopted by OODB in modeling complex objects (or alternatively called composite objects).

2.3.2 Generalization

Another important and powerful data abstration by SDB is generalization—-an idea for modeling "is-subtype-of" (or alternatively "is-subclass-of") relationship. An associated concept here is *inheritance*: inheritance of memberships up the hierarchy, and attributes/relationships down the hierarchy. Again, this feature has been adopted into OODB, and has been extended to include inheritance on methods.

2.3.3 Extensibility

With OODB, one can define new types/classes from predefined ones. Such a system embodies an extensible one. This feature has had some impact to SDB. Despite early SDBs were as non-extensible as traditional databases, we are now seeing later SDBs have similar capabilities.

2.3.4 Computational Completeness

Another impact from OODB to SDB is on computation aspect. With OODB, one can express any computable function supported by the programming language. This has inspired SDBs to built some language interfaces through which external functions can be encorporated into DML.

2.4 Distinct Features in SDB and OODB

Being developed from different roots and application domains, SDB and OODB also embody substantial differences in supporting distinct features, which reflects the fact that SDB emphasizes on semantic

expressiveness, while OODB emphasizes on behavioral power.

2.4.1 Constraints and Exception Handling

In SDB, a large number of constraints can be represented explicitly, and exceptional information can be accommodated. This is in contrast with OODB, where only very limited constraints and exceptions are supported.

2.4.2 Recursiveness

Another powerful feature supported by SDB is on the recursive applications of its constructs (e.g., aggregation and grouping [7]). This is again not typical in OODB.

2.4.3 Uniformity

Finally, SDB emphasize uniform treatment of all information. Thus, descriptive information about objects (referred as meta-data), is conceptually represented and treated in the same way as specific "fact" objects. This feature seems to be missing or overlooked in OODB.

2.4.4 Behavior Encapsulation

On the other hand, OODB also offers features which are absent in SDB. The most obvious one is the encapsulation of operations (called "methods") within types. This complements on the complete definition of the types, and extends the ways for interacting with other types, deriving new objects, etc.

2.4.5 Overriding, Overloading and Late Binding

Related to the method embedding is the concepts of method overriding (by subtypes), overloading and late binding. Again, such concepts are absent from SDB.

2.5 Other Features Assumed/Needed

To be a complete database system, other features which can be ascribed to traditional databases will have to be added. These include: persistence, query facility, secondary storage management, transaction, concurrency, and recovery. We omit the discussions of them since they are not model specific for SDB and OODB.

3 Summary, Discussion and Conclusion

We have examined 3 categories of features supported by SDBs and OODBs. These can be summarized by a table as follows:

Data Model Features		SDB	OODB
Common/Similar Ones	Object-id	Yes	Yes
	Types/Classes	Yes	Yes
	Relationships	Yes	Yes
Adopted/Influenced	Aggregation	Yes	Adopted
	Generalization	Yes	Adopted
	Extensibility	Influenced	Yes
	Full Computation	Influenced	Yes
Distinct Features	Constraints and Exceptions	Yes	No
	Recursions	Yes	No
	Uniformity	Yes	No
	Behavior Encapsulation	No	Yes
	Over-riding/-loading	No	Yes

From the above table, we see that SDB and OODB are closely related to (and indeed, inseparable from) each other. Not only they have influenced each other, but also they resemble a similar paradigm in modeling and structuring the objects, which reflects some common expectations from the users of the next generation databases. By looking at the above three categories of features, we can also infer that SDB and OODB are by large compatible for integration, and indeed, are desired to be integrated as they can complement each other. Such an integration shall promote a complete object power and paradigm, which may be tentatively called as the "objective databases" (or simply, "object databases") (ODB). In conclusion, we propose the formula: ODB = SDB + OODB for the definition of an object database, emphasizing the importance of encorporating both structural and behavioral facilities into the next-generation databases.

References

[1] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 430–440. ACM, 1987.

[2] M. Atkinson, F. Bancilhon, D. DeWitt, and S. Zdonik. The object-oriented database system manifesto. In *The 1st Int'l Conference on Deductive, Object-Oriented Databases*, pages 40–57, Kyoto, Japan, 1989.

[3] P.P Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[4] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.

[5] K. Dittrich. Object-oriented databases: The notions and the issues. In *Proc. of the Int'l Workshop on Object-Oriented Database Systems*. IEEE Computer Science Press, 1986.

[6] M. Hammer and D. McLeod. Database description with sdm: A semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[7] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[8] W. Kim and H. Chou. Versions of schema for object-oriented database systems. In *Proceedings of the International Conference on Very Large Databases*, September 1988.

[9] W. Kim and F.H. (Editor) Lochovsky. *Object-Oriented Concepts, Databases, and Applications.* ACM Press, 1989.

[10] R. King. My cat is object-oriented. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications.* ACM Press, 1989.

[11] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented dbms. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 472–482. ACM, 1986.

[12] J. Peckham and F. Maryanski. Semantic database models. *ACM Computing Surveys*, 20(3):153–189, September 1988.

[13] S. Zdonik and D. (Editor) Maier. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, 1989.

Towards an Optimum Language Data Model

Ed Lowry
Digital Equipment Corporation
200 Forest St, Marlboro MA 01752

Abstract:

Maximizing the satisfaction of a value in an engineering design is usually limited by tradeoffs in which other values become unacceptably sacrificed. In a few cases, however, the maximization is limited by a boundary between what is mathematically possible and what is not. Round wheels, vertical pillars, and binary memory elements are examples of optimum engineering structures which result from such mathematical limits. It is proposed that optimum characteristics of a language data model result similarly by minimizing the variety of primitive data objects, the complexity of those objects, and the number of objects needed to represent data states. Reducing these measures is needed to combine both rich data structure and powerful operations in one language. The minimizations lead to a narrow range of designs for language semantics in which the potential advantages of specialization is small compared with the advantages of commonality. Universal language for support of technical literacy appears to be an appropriate scope of generality in language design.

Language Data Models

Any formal language references subject matter which has a well defined class of permissible structures. That class of structures can be referred to as the data model of the language. The data model is the most basic part a language in the sense that other parts of the language may be changed without disturbing the data model while changes to the data model force changes to most other parts of the language. The mathematical character of data models and the need for simplicity in their design limits the variety of reasonable designs. Reasons are given below why improvements in the design of data models lead to a narrow cluster of design choices within which the scope for specialization is small.

Almost contradictory requirements

In designing any representational system (musical scores, i[illegible] sculpture, etc) there is a need for both:

- accuracy of representation

and

- ease of working with the representations.

This article expresses the author's views, not those of any organization.

For symbolic systems these requirements tend to conflict. Representational accuracy is usually achieved by using many structural primitives, while ease of operation seems to require restriction to very few. In computer languages this conflict is illustrated by a dichotomy between:

- structurally expressive languages (Ada, PL/1, etc using MANY primitive types of data object)

and

- functionally expressive languages (APL, Relational DB, etc using very FEW primitive types).

When applications are developed, the conceptual structures of the application must be represented using the available data structures of the language. For current structurally expressive languages, the data structures represent the conceptual structures in a fairly accurate way but the programs are complicated by the lack of powerful operations. When applications are developed in current functionally expressive languages, complications are added as a result of the need to represent the conceptual structures using data structures which cannot represent them so accurately. The complications occur any time the problem domain has more than a small amount of structural richness.

Extending current languages

Efforts to add to the data structure richness of an existing functionally expressive language has tended to complicate the language excessively. The initial complexity of the structurally expressive languages has tended to discourage attempts to add significantly to their functional expressiveness.

Three hypotheses on boundaries to data model improvement

Reasons are given why optimizing the simplicity of expression in a formal language (for non-trivial structures) is limited not by a tradeoff but by a boundary on what is mathematically possible. The major hypothesis is that the optimum data model design in this sense results when the primitive data structures are as simple as possible. This hypothesis is subdivided into three parts in order to clarify its meaning, provide a process for verification, and to express reasons for its plausibility.

1. Simplicity of expression is maximized by minimizing the variety of primitive data types.

Functional expressiveness includes the ability to provide nested expressions in which the results of executing subexpressions are data aggregates which can be used as operands of outer expressions. Keeping the variety of primitive data structures small increases the likelihood that a data aggregate resulting from one expression execution can be used as the operand of others without expanding the variety of expression types. Empirically, languages which provide this capability effectively are limited to one basic type of data structure.

To verify the need for a minimum variety of primitive data types we can exhibit one (or more) languages with a very small variety of primitives and a high simplicity of expression. Continuing good faith efforts which fail to produce anything with fewer types of primitive that works and fail to produce anything with more types of primitive and greater simplicity of expression, would increasingly confirm the hypothesis. The KEEP language discussed below is exhibited in the expectation that such efforts will fail or provide insight into how to design a language for which the efforts will fail.

2. Simplicity of expression is maximized by using primitives composable from the simplest possible structures.

The first hypothesis can be independently confirmed. Its plausibility contributes to the plausibility of this one. Representational accuracy includes the ability to represent a wide variety of structures with little distortion. These include both very simple and very complex structures. The first hypothesis restricts the language to represent all structures using only an extremely small variety of primitive structures. If the chosen primitives are complex, they can only be composed to give structures that are more complex and simple structures will not be accurately represented. This suggests that simpler is better in choosing a small variety of primitive structures to accurately represent a wide variety of data strucures. A lower bound on the structural complexity of acceptable primitives is that they must provide a way to represent relationships between objects.

Confirming this hypothesis can be done in the same way. We exhibit a language (KEEP) with extremely simple primitives and look for ways to design any effective language with simpler primitives and look for ways to provide a language with greater expressive economy but more complex primitives. Continuing failure to find those would provide increasing confirmation.

3. Expressive economy is optimized by constraining the composition of the (extremely simple) primitives in a way that minimizes the number of objects used to represent data states.

Assuming that the first two hypotheses are confirmed, then confirmation of this hypothesis could further guide data model design. The minimization will occur in languages for which there is very little constraint on how simple structures may be composed to represent complex objects. This flexibility increases the likelihood that data structures closely matching the conceptual structures can be formed. It rules out data models which can be described as composed from extremely simple objects but only in costrained ways so that the primitives may be more accurately described as something more complex (such as binary tree elements in pure Lisp, or sets of tuples in Relational Algebra).

The style of confirmation can be the same. We exhibit a language (KEEP again) which has a high degree of flexibility, and look for ways to design a language where data structures can be more economically represented or less economically represented with greater expressive economy.

KEEP

The KEEP language provides a number of empirical observations which suggest and support the above hypotheses and those listed further on:

- KEEP does provide for both structural and functional expressiveness as effectively as any language known to the author which is biased toward one or the other.

- KEEP and other languages tending toward its capability use only an extremely small number of types of primitive object which have extremely simple internal structures. The objects are used economically, in the sense that the number used to represent data states is about as low as possible.

- The KEEP language is simple compared with languages like PL/1, Ada, Common Lisp. The definition is about one quarter their size. Small subsets can be defined.

- A language with the semantics of KEEP can (but need not) have a syntax which gives it a very intuitive natural language style.

These observations suggest that there is little need to compromise or specialize in the following possible areas of tradeoff:
- structural richness or functional power.
- language simplicity versus comprehensiveness.
- naturalness of expression for any of the above.

Engineering optima based on mathematical limits

There are only a limited number of engineering solutions where increasing some value is stopped by the mathematical impossibility of going further. They include:

- Circular is the optimum crosssection for wheels and rotating bearings. (Minimizes energy transfer.)
- Binary is the optimum radix for digital memory elements.
- Tubular is the optimum shape for pipes carrying pressurized fluid.
- Cylindrical is the optimum cavity shape for propelling pistons or bullets.
- Horizontal is optimum for axles and floors.
- Vertical is optimum for loaded walls, pillars, and door hinges.
- Monoplane and bilateral symmetry is optimum for fixed wing aircraft wings (after the structure is sound).
- Flat is optimum for wall mirrors and broad saw blades.
- Paraboloidal is optimum for telescopic mirrors.
- Helix is optimum for bolt threads and longtitudinal springs.
- Circular is the optimum orbit for particle accelerators.
- Uniform is the optimum distribution of tensile strength for cables and chains.

This kind of engineering solution generally:
- Gets near universal acceptance.
- Endures indefinitely. The extremum is easily approached (in many cases) and impossible to exceed.
- Offers little scope for compromise.
- Arises rarely but forcefully.

The general acceptance of these solutions is often a convenience. In the area of language design, comparable acceptance could be much more valuable because the effectiveness of communication is greatly enhanced when all participants use the same language.

In each of these cases there are many interacting values which affect the total engineering design. However, the interaction between the values did not prevent one of the values from being stopped by a mathematical limit rather than a tradeoff over a wide range of conditions. A marginal case is the choice of monoplane. In the early development of aircraft, the tradeoff between aerodynamic efficiency and achievable structural soundness favored a biplane design. Another marginal case arises in bridge design. If the span is so large that shear forces and bending moments become excessive, then purely tensile cables are preferred for the main spanning members. The near parabolic shape of the suspension bridge cable is not so much an optimum choice as the result of choosing flexible cable.

Secondary hypotheses

A series of secondary hypotheses is presented in the hope that their confirmation or refutation will clarify appropriate directions for improving computer language and perhaps a wide range of technical communication. The breakdown into many hypotheses is intended to help focus the evaluation of evidence on specific issues.

The hypotheses are sequenced so the early ones are more plausible. Refutation of one hypothesis will also tend to refute later ones. The later hypotheses are more vulnerable to refutation, but have more significance if confirmed. The earlier hypotheses become more significant if the later ones are refuted, so it is probably most productive to evaluate the later ones first.

Evidence supporting or refuting each will usually be derived from comparisons of complexity of well written system descriptions which have been translated into languages with different characteristics. In most cases confirmation will result from the observation that substantial good faith efforts cannot exhibit counterexamples. The main problems in providing confirmation will probably be in encouraging such efforts and identifying them as such. The meaning of each could be more precisely stated in terms of the classes of phenomena that never occur.

The hypotheses apply only to language used to describe systems with non-trivial structure. It is plausible that the value of language specialization will decrease as the structure of domains of discourse become richer.

So far the evidence seems to support all of the following:

4. Structural and functional expressiveness can be combined in a single formal language without interfering with each other.

5. No significant technical advantages can be designed into a language not combining structural and functional expressiveness which cannot be made available in one that does.

6. Structural and functional expressiveness can be combined in a single formal language which is restricted to economical use of extremely simple primitive data objects.

7. Effectively combining structural and functional expressiveness in a single formal language REQUIRES a design which is restricted to economical use of extremely simple primitive data objects.

8. The above requirement restricts the range of effective designs so that potential advantages of data model specialization are substantially less than the advantages of a single standard data model.

9. The main features of high quality data model design, are determined by mathematical boundaries which abruptly limit simplification of the primitive data objects.

10. The above abrupt limits restrict the range of effective designs so that potential advantages of data model specialization are orders of magnitude less than the advantages of a single standard data model.

11. The above abrupt limits restrict the range of high quality language design choices so that potential advantages of language semantics specialization are substantially less than the advantages of a single standard language semantics for definition of structurally non-trivial formal systems. (Within a relatively simple or unchanging formal system, language may be less formal and more specialized.)

12. A language which combines structural and functional expressiveness can have enough clarity and precision to play a major role in technical communication and technical education.

13. There are no technical obstacles to the development of a universal language for supporting technical literacy.

A NEUTRAL OBJECT-ORIENTED DATA MODEL

Robert Marcus
Boeing Advanced Technology Center
P.O. Box 24346, MS 7L-64
Seattle, Wa 98124
rmarcus@atc.boeing.com

There are an increasing number of object-oriented databases and programming languages becoming available in industry. Each product comes with a different model of object-oriented structure and data. In addition, it is necessary to integrate most of these products with previously existing applications and databases. In order for the object-oriented paradigm to achieve major breakthroughs in large companies, it is necessary to supply a mechanism for integrating legacy systems and multiple object-oriented tools. This mechanism must also provide a methodology for migrating away from older applications and databases without disrupting the operation of the organization.

One of the necessary conditions for this type of integration and migration mechanism is a structured way of insulating applications from physical databases. A methodology for accomplishing this that has achieved wide acceptance is the three-schema architecture. The three schemas are:

a) Top level application views of the data.
b) A neutral data model conceptual view of the data.
c) Bottom level actual database views of the data.

The neutral data model is the key component of the three-schema architecture. Mappings are built from this level to application views and actual database schemas. In this way, applications are insulated from the underlying database structure and location. Currently commercially available three-schema architectures use simple entity-relation or relational schema as their neutral data model. These representations are inadequate for as a communication medium between object-oriented applications and data.

In order to utilize the three-schema architecture in future object-oriented environments, it is necessary to have a neutral object-oriented data model that is rich enough to capture the semantics of the majority of object-oriented applications and databases. At the same time, this neutral model must be elegant enough to facilitate easy mapping from and to other data models. Another requirement growing out of the need to integrate with legacy systems is that the neutral data model provide an easy mapping to relational schema.

I believe that the most important standard for the future of object-oriented methods is this neutral object-oriented data model that can serve as the foundation of three-schema architectures. A standard in this area will not restrict vendor activities in object-oriented databases or programming languages while provide end-users with a method of integrating new tools into existing environments.

Due to Boeing constraints, I will not be able to submit a paper in a timely fashion for the OODBTG meeting in Atlantic City. I have enclosed a paper that appeared recently in the journal, "New Generation Computing" which is very close to my vision of a neutral obejct-oriented data model.

OODB Standardization

Roger Osborn
Concurrent Computer Corporation
Slough, England

1 Introduction

This position paper describes the features of an Object Oriented DBMS and their appropriateness for standardization. It is based upon work being undertaken by Concurrent Computer Corporation.

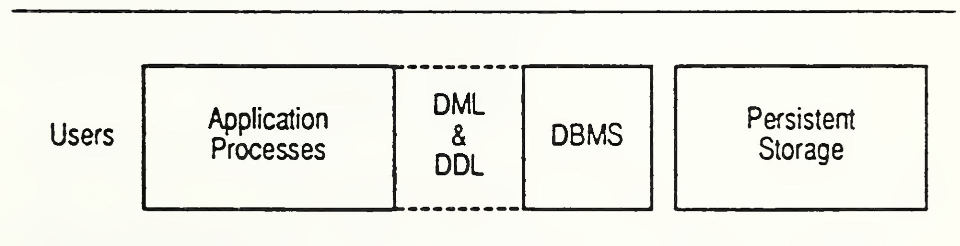
2 Purpose of a Database Standard

2.1 Database

A database provides facilities for the management of information. It provides access, update and persistent storage of information. It maintains the consistency or integrity and the security of information. A database is supported by a DBMS (Data Base Management System).

Typically a database consists of one or more schemas, which define the entities in the database, and the information holding entities themselves. It is interfaced by, a data definition language with which a schema is created and maintained, and a data manipulation language with which information is accessed and updated.

The following diagram illustrates an environment model of DBMS use.



2.2 Standardization

It is towards the definition of data definition and manipulation languages for specific types of entity that database standardization has primarily addressed itself.

There are a number of benefits that are obtained if DBMS products conform to a common standard.

- **Mixed systems interworking.** Products that conform to the same standard or rely on a standard interface to other products can be used together to support an application;
- **Exchange and re-usability.** Transfer of information between standard products and replacement of one standard product by another are both facilitated;
- **Confidence.** The fact that a product conforms to a standard gives a prospective user confidence that the product provides well understood and accepted functionality appropriate to its particular problem domain;
- **Familiarity.** The knowledge and skills acquired by users of one standard product are easily applied to other products conforming to the same standard.

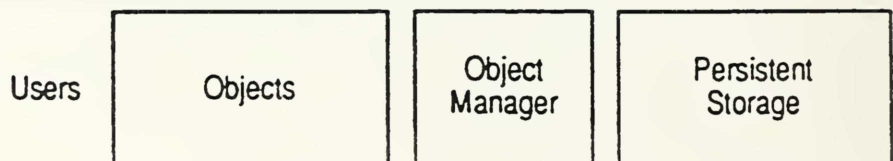
3 Object Oriented Database

3.1 Overview

An object encapsulates some state and behaviour. An object oriented database supports both persistent and non-persistent objects. It is through objects that the database features of information storage, access and update are provided, and information consistency and security are maintained.

Applications are supported as the behaviours of objects, application processing consisting of the processing of invocations of objects. Thus there is a single model for both application and information management activity. An object oriented DBMS therefore provides facilities both for information management and application processing. (OODB and OODBMS are perhaps misnomers as they suggest a separation, between application processing and information management, that should not exist in an object oriented environment.)

The following diagram illustrates an environment model of object oriented DBMS use.



The object manager provides an object invocation service and other basic services, including security and consistency, used by all objects.

An object oriented database can co-exist with and be used by a traditional application system provided that facilities are supported that enable its objects to be invoked by application processes.

3.2 Standardization

To achieve the benefits of standardization for an object oriented DBMS, both topics covered by the data definition and manipulation languages for non OODBMS's and topics raised by the support for application processing within the DBMS, should be addressed. The scope for standardization is therefore extensive. It is suggested that the following items are considered:

- the object model itself.
- object specification and any necessary language support.
- the process model with particular concern for the semantics of object invocation.
- information access and query mechanisms or language.
- information consistency and security mechanisms.
- specific object types.

Standardization should not of course compromise the fundamental concepts of object orientation, neither should it limit the flexibility and power of the approach.

4 Object Oriented Database Features

The following sections describe the features of an OODBMS. The features could be the basis of a reference model for OODBMS standardization.

4.1 An Object Model

4.1.1 Structure

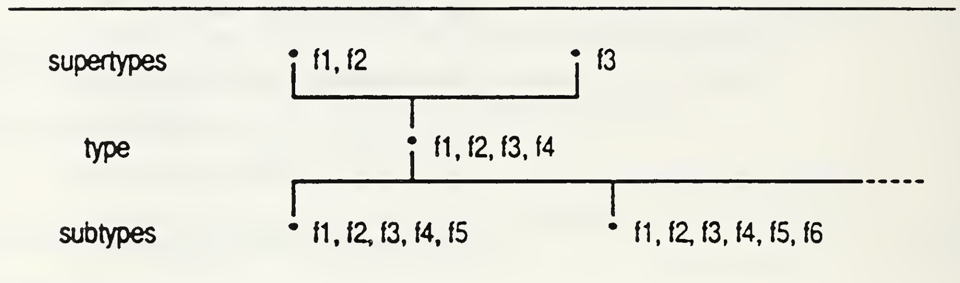
An object has an identity and encapsulates information and behaviours. There is a strong separation between the external interface or view of an object and its internal implementation. Internally it has a state (or instance variables) and code bodies (methods) which act upon that state, whilst externally it supports a number of functions and attributes. (An attribute is a function with a single parameter, the output value of the attribute). A method corresponds to a function (or attribute) so that, when an object is invoked and a function selected, the corresponding method is executed. The invoker is ignorant of how the invocation is

processed, and has no knowledge of the internals of the object, the state and methods being completely invisible and inaccessible to it.

4.1.2 Type

An object's type defines its external view. All objects of the same type support the same set of functions and attributes. However because two objects support the same function, does not imply that they execute an identical method to process an invocation of that function.

Object types form a lattice where an object of a given type supports all of the functions and attributes that objects of its supertypes support, plus any additional functions and attributes specified with the type. A type is said to inherit from its supertypes.



A type is an object in its own right. It supports functions (and attributes) that provide information on the definition of the type. Information such as descriptions of the functions and attributes, and their parameters, that are supported by objects of the type, is provided.

4.1.3 Implementation

As well as a type, an object has an implementation. Whereas the type defines the external view of an object, the implementation defines its internal view. Thus the implementation describes the code bodies of the methods and a template or structure for the instance variables or state. An object is an instance of an implementation. An implementation corresponds to only one type. Thus all objects, that are instances of the same implementation, have the same type. However a type can correspond to more than one implementation.

Inheritance between implementations is possible, so that an implementation inherits the instance variable templates and methods from its super implementations. The inherited templates can be extended to incorporate locations for new instance variables. New methods are specified as necessary for the corresponding type. An inherited method can be replaced by a new method thus giving a new implementation for a function. Inherited instance variables cannot be directly accessed by new methods as this could lead to unpredictable changes to the behaviour of inherited methods. Like a type, an implementation is an object in its own right.

In order that the power of object encapsulation is not compromised, any standardization of implementation should not preclude alternate completely different approaches.

4.1.4 Object Creation

An implementation supports a function 'create instance.' Invocation of this function causes a new object, that is an instance of the implementation, to be created.

4.1.4.1 Type and Implementation Creation

As a type is an object in its own right it has a type 'type-type' and an implementation 'type-implementation.' Invocation of create-instance on the object 'type-implementation' creates a new type.

Similarly for implementation there is type 'implementation-type' and implementation 'implementation-implementation.' Invocation of create-instance on 'implementation-implementation' creates a new implementation.

4.2 Processing Model

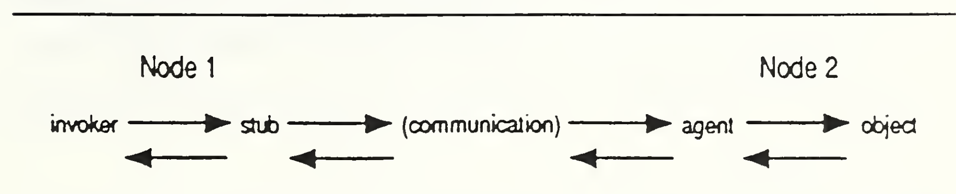
Subject to any underlying system restrictions and to any algorithmic limitations of its methods, an object may support the concurrent invocation of any number of its functions.

Processing activity in an OODB can be visualized as the execution of threads of nested invocations, that are generated as the method of one object invokes another object and so on. Within a thread, invocations are processed synchronously, so that the execution of a method is blocked until its invocation of an object returns. However this does not preclude asynchronous activity. A method can use multiple threads of execution in the processing of an invocation, and it can initiate independent threads of execution.

If a method fails to complete its execution for any reason, an exception is raised and the nest of invocations is unstacked up to a point where the exception can be handled.

In a distributed system, multiple object managers and their objects co-operate and interact to support a distributed database and to provide location transparency for object invocation.

An object at one node in a distributed system can invoke an object at another node. The mechanisms of the inter-communication are encapsulated by a stub object, that represents the remote object at the invokers node, and an agent object, that represents the invoker at the remote node.



4.3 Object Specification

4.3.1 Invocation Parameters and Instance Variables

Parameters for functions can be input or output and can be typed object references or primitive types such as integer, real, array, string, structure, etc., whatever is supported. The type, of the actual object referenced by a typed object reference, can be either the specified type or any subtype of it.

Instance variables can be either typed object references or primitive types.

4.3.2 Languages

In principle, a method can be specified in any programming language. To fully achieve this, it is necessary to extend some languages to enable a method to access the instance variables and the invocation parameters as well as any local variables that it might declare. It may also be necessary to extend languages so that they support an object invocation statement.

As the methods of an object are invisible to its invokers, mixed language systems, where the code of the invoker is written in one language and the executed method is written in another, are very possible. This can necessitate translation of invocation parameters, possibly using a notation such as ASN1 in the communication of an invocation from an invoker to an invokee.

4.4 Information Management

4.4.1 Persistence

The state of a persistent object is maintained in a persistent object store. When a persistent object is invoked, the object manager ensures that the state is loaded or mapped into memory. When a method updates the state, it is the methods responsibility to ensure that state changes are copied to the persistent object store. It might do this explicitly by calling an object manager 'save' function, or it might rely on memory management hardware and software to detect the updates and copy them back.

4.4.2 Access

4.4.2.1 Primary Access

An object is accessed via the object manager which binds an invocation to an object based upon an object reference that uniquely identifies the object. The object reference either refers to the objects previously fixed up in memory location, or it contains the object's unique identifier. An object reference is obtained from another object as an input or output invocation parameter. It can be held as an instance variable.

4.4.2.2 Collections

An object could contain a collection, that is a list or other structure, of object references, and support functions which return selections from the collection. Selection conditions, input with a function invocation, could contain expressions based upon the common attributes of the objects in the collection. A collection could maintain indexes based upon attribute values with which to speed its processing of selections.

If required such a collection could support the abstraction of a relation, its functions supporting a relational query language based on the types of the objects in the collection.

4.4.2.3 Attribute Relationship Based Navigation

An object has a type relationship with its type object and an implementation relationship with its implementation object. A type has a supertype relationship with its immediate supertypes and a subtype relationship with its immediate subtypes. A specific association or aggregate (sub component) relationship between two objects is represented by the attribute of one object referring to the other object.

It is possible to navigate this network of inter object relationships to search for required objects. It might be possible to devise a language with which to express the criteria for such a search.

4.4.3 Consistency

4.4.3.1 Transactions

Transactions enable information changes, that require multiple updates to one or more objects, to be made without any loss of consistency. The object manager supports a transaction manager that is invoked by methods, when they wish to start, commit or fail a transaction. All updates within the context of a transaction are subject to the successful committal of the transaction, otherwise they are rolled back. Each object is responsible for ensuring that its updates adhere to the correct procedures for a transaction. Either an object explicitly supports 'prepare to commit', 'commit' and 'roll back' idempotent functions that are invoked by the transaction manager on transaction commit or roll back, or it relies on default mechanisms provided transparently by the object manager in the implementation of its 'save' function or update handling service.

Updated instance variables are locked against access by methods executing other than in the context of the same transaction, until the transaction completes. Implementation and understanding are simplified if all of an object's instance variables are locked, thus making the effective granularity of the update to be an object rather than part of an object. However for large objects this may not always be acceptable.

Nested transactions enable a method to control the consistency of the updates, that is causes, and to manage failures, independently from any considerations of the transaction context in which its object was invoked. A nested transaction can be rolled back independently of its containing transaction. On committal its locks and updates are inherited by its containing transaction. Updates are only irreversibly committed when the outermost transaction commits.

In mixed or distributed systems, when a transaction involves objects subject to different transaction managers, the transaction managers use an appropriate protocol to co-operate in the support of a distributed transaction. Standardization of inter transaction manager protocol for distributed transaction support should be consistent with other transaction standardization activities (ISO 9805 and 9804).

4.4.3.2 Semantic Integrity

Attributes and instance variables can represent semantic relationships between objects. Methods can be coded so that, if their action causes a change that affects a semantic relationship, then they trigger themselves to invoke the object concerned to inform it of the change. Thus the effects of semantic changes can be propagated through objects and semantic integrity can be maintained.

4.4.4 Security

An object is a unit of appropriate granularity for security control. An invoker has the rights or not to invoke all or a subset of the functions and attributes supported by an object. The object manager mediates the rights of the invoker to the object. Capabilities, which are protected unforgeable object references, provide an efficient and flexible mechanism upon which to base security. If the invoker does not supply a valid capability with an invocation, then the invocation is rejected. A capability can be passed as an object reference parameter between objects, and can be held as an instance variable.

When an object is created, the invoker is returned an object reference containing a capability granting all rights to the object. Copies of this capability, with all or a subset of the rights, can be made and passed to other objects when required, thus granting them rights to the object. Capabilities can include or not the right to make further copies. Capabilities can be persistent or transient.

4.5 Specific Object Types

There is much scope for the standardization of specific object types, both for types which extend the basic features of an OODBMS and for higher level types.

Collection object types and an associated query language could be specified.

Types of objects specific to particular application domains could be specified. These would make possible the ability to construct applications from libraries of standard types.

5 Conclusion

This paper has presented a position on OODBMS standardization. The features of an OODBMS which could be a basis for such standardization were briefly introduced. OODBMS standardization is an extensive topic which unifies many aspects of database, language and processing models. There is much to do.

X3/SPARC/DBSSG OODB Task Force

Group Workshop

Standardization of Object-Oriented Database Systems

May 22, 1990

Position Paper

Daniel O. Sanderson

Digital Equipment Corporation

1175 Chapel Hills Drive

Colorado Springs, CO 80920

1 Introduction

An object-oriented development system is made up of many components, among them:

- An object-oriented language
- A method dispatching mechanism
- A peristent object store
- Browsing tools
- The underlying hardware

With these components in place, applications can be written that take advantage of the many benefits of ''object-orientedness''.

While there is a need to standardize many of these components, we must not ignore the need for accompanying methodologies, techniques, and tools that support the design and development of applications within an object-oriented system.

In traditional application development, a high-quality language and database system do not necessarily result in high-quality applications. Likewise, it takes more than a high-quality object-oriented language and database system to develop good object-oriented applications. Even if a standard object model, OODB

interface, and language are adopted, they still do not adequately address all the problems of designing and developing applications, especially large ones. This is because the process of developing software is a complex one, no matter what implementation alternative is used, and, as Brooks noted, there is ''no silver bullet,'' not even an object-oriented one! [BROOKS]

So, what is needed, in addition to standardizing terms and models, is a formal methodology that takes advantage of the object model. In this paper, I will look at traditional development methodologies and propose some requirements for a new methodology that is based on object-oriented concepts.

In general, a *methodology* is a prescription to accomplish a task or series of tasks. It has three components [BUBENKO]:

1. A statement of the problem or requirements

2. A process (usually iterative) that leads to a solution of the problem

3. A theoretical proof or empirical evidence that shows that the process results in a correct solution

A *technique* is a specific way in which a methodology, or part of it, is applied, and a *tool* provides automatic or semi-automatic support to practice a technique.

While formal methodologies and techniques for traditional application development have been around for quite some time, along with a number of tools, only a few such methodologies exist for applications built within purely object-oriented environments.

2 Traditional Methodologies

Structured analysis and design methodologies, such as those set forth by DeMarco [DEMARCO] and Yourdon [YOURDON], have been in use for many years. Using these methodologies has been shown to increase productivity and decrease errors, yet many software development organizations still do not use them. This may be due, not to deficiencies in the methodologies, but to confusion over which one is best, how the methodologies can work together, and the lack of understanding as to what they can and cannot do. Some also view the techniques that have risen out of these methodologies as being too narrow and applicable only to certain classes of applications, such as MIS applications.

Attempts to apply structured analysis techniques to object-oriented systems have largely failed, primarily because the world-view models on which they are based are very different. Structured analysis and design focus on the functional aspects of a system, such as processes, decision trees, and sequential algorithms, while object-oriented techniques attempt to model the world in terms of objects and methods.

Database design methodologies, such as Entity-Relationship (E-R) modeling [CHEN] and Binary-Relationship modeling [MARK], have also been in use for a number of years; however, these are practiced even less than structured analysis. A number of extensions to these modeling techniques, especially to E-R modeling, have been proposed, some even adding so-called object-oriented extensions [TEOREY][SMITH]. For example, E-R has been extended to include notions of inheritance and subtyping. Such extensions, though, address only the structural modeling of objects. Without the ability to model both object behavior *and* object structure within the same methodology, extensions to existing database design methodologies will remain just that: extensions.

The weaknesses of the traditional design methodologies can be summarized as follows:

- Traditional methodologies do not adequately address separation of specification and implementation.
- The various traditional methdologies and sub-methodologies are not well integrated.
- To model all aspects of an application, the designer or analyst must model structure and behavior separately; this does not correspond closely to the way that things actually are in the real world.

3 Requirements for an Object-Oriented Methodology

Structured design and data modeling methodologies have attempted to improve the usefulness and reliability of systems built on third- and fourth-generation languages and relational DBMSs by providing a formal process for dealing with parts of a large system in a unified way. These methodologies have been successful to a certain extent, but they also suffer from a fair amount of disjointness and lack of natural mapping to the problem being solved.

The object model provides us with a unique opportunity to ''start from scratch''

in coming up with a fresh approach to building new applications that are based on object-oriented languages and databases. In the object-oriented space, we need a good methodology (accompanied by techniques and tools) that will make object-oriented systems useful and attractive to implementors of large applications.

Such a methodology will no doubt be difficult to discover. Most likely, it will comprise a number of sub-methodologies, each of which addresses a part of the development life cyle. This is similar to the way that data-flow diagrams and decision trees are both part of a larger methodology called structured analysis and design. However, we can learn from the traditional methodologies, and the disjointedness between them, and develop a consistent methodology that addresses all aspects of the application life cycle.

My intention is not to propose such a methodology but to set forth a preliminary list of requirements:

Language Independence

The methodology should be language independent and be based on a standard object-based model, such as that being developed by the OODB Task Group. For example, it should be possible to design an application regardless of whether it is implemented in C++, Smalltalk, or some other object-oriented language.

Life Cycle Support

The methodology should address all aspects of the development life cycle, including requirements definition, architectural and detailed design, implementation, and testing. For example, system-level requirements should be expressible in terms of objects.

Tool Automation

The methodology should be formal enough so that automated tools can be built that assist designers in creating, tracking, and verifying designs. For example, the methodology should prescribe the rules for consistency between parts of a design that can be automatically checked by a tool.

Policy-Free

The methodology should allow for individual organizations to customize it with their own policies and standards. For example, an organization may want to

enforce certain naming conventions by customizing the methodology to include such rules.

Consistency

The sub-methodologies should be consistent with each other and allow for natural transition between them. For example, all sub-methodologies should make consistent use of terms and symbols.

Modeling Features

The methodology should support the modeling of all aspects of an application, including at least the following:

- Structure and behavior of object types
- Transaction semantics
- Physical and logical distribution of objects
- Security of objects
- Complex objects types, such as object containment hierarchies
- Inheritance
- Active objects and triggers
- Persistent object storage and retrieval
- Requirements definition

4 Conclusion

The advent of a standard, unified object model gives us an opportunity to improve the quality of object-oriented applications and decrease the time it takes to produce them. But good languages, reliable object engines, and fast hardware are not sufficient. Although these things give us a great amount of power, we need to apply formal methodologies in order to harness this power. Traditional methodologies and techniques are insufficient to take real advantage of object-oriented databases and languages.

A new methodology, based entirely on object-oriented concepts, must be defined

that will increase productivity of our precious engineering resources and maximize the benefits of the object model. Out of such a methodology, we can develop practical techniques and build automated tools that help us to follow the methodology and avoid costly design and implementation errors.

5 References

[BROOKS] Brooks, F.P. "No Silver Bullet: Essence and Accidents of Software Engineering," *Information Processing '86*, ISBN No. 0-444-70077-3, H.J. Kupler, ed.; Elsevier Science Publishers B.V. (North-Holland) IFIP 1986.

[BUBENKO] Bubenko, J.A., and S. B. Yao, "Data Base Design Tools," CH1389-6/78/0000-0002$00.75 (c) 1978 IEEE.

[CHEN] Chen, P.P. "The Entity-Relationship Model -- Toward a Unified View of Data," *ACM Transactions on Database Systems*, Vol. 1, No. 1, March 1976.

[DEMARCO] DeMarco, T. *Structured Analysis and System Specification*, New York: Yourdon Press, 1978.

[MARK] Mark, L. "What is the Binary Relationship Approach?", In Davis (Ed.) *Entity-Relationship Approach to Software Engineering*, North-Holland, 1983.

[SMITH] Smith, J.M., and D.C.P. Smith, "Database Abstractions: Aggregation and Generalization," *ACM Transactions on Database Systems*, Vol 2, No. 2, June 1977.

[TEOREY] Teorey, T.J., D. Yang, and J.P. Fry, "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model," *Computing Surveys*, Vol. 18, No. 2, June 1986.

[YOURDON] Yourdon, E., and L.L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 2nd ed. Englewood Cliffs, N.J.: Prentice-Hall, 1979.

Donald B. Sanderson
Rensselaer Polytechnic Institute
Design Research Center

OODBTG Workshop on Standards Position Paper

March, 30 1990

One of the most commonly observed phrases in the introduction of a paper about object-oriented database systems (OOODBMS) is a disclaimer that there currently exists no generally agreed upon definition of an OODBMS, but that the prototype proposed in that paper is, in the authors' opinions, an example of one. Even the basic data model issues in an object-oriented environment are subject to different interpretations. As an example consider the diversity of data models employed. The data model used in ROSE [Har] includes a specialized set of aggregation and generalization abstractions, implemented as extensions to C++ and Objective-C, and which are made persistent. The GemStone [Bre] data model is based on Smalltalk, and makes it possible to save any valid object to disk. The IRIS [Wil] system extends the SQL language to define objects to be stored and managed by the system. With no clear consensus in the DBMS community on the definition of an OODBMS, it seems premature to attempt to define a set of standards. The SQL and DBTG standards were successful due to the fact that they served mainly to codify and formalize definitions that were already widely accepted.

This is not to say that we should not be discussing the issues that will be required for standards. This will in fact be

advantageous when there is enough consensus for a standard to be created. The main focus of these discussions should be in areas which do not limit the scope of OODBMS research, but which try to pr vide a functionality that will make eventual integration of object-oriented systems possible. The key issue in my opinion is that of data exchange standards.

Data Exchange between existing DBMSs is a difficult problem, even with the current exchange standards. The systems which are most promising are those that work by translating queries from one DBMS language to the language of another, and then use this query to materialize the data to be passed back to the first system. This works well when there is a similar query language for both systems, but is not applicable to data sharing between DBMSs with different underlying models. It also does not allow for exchange of data between systems that do not integrate a query language with their data descriptions such as is the case for EDIF and IGES.

Early work on the data exchange problem will aid both in the development of standards, and in the exploration of OODBMS. If researchers have the ability to share both raw data, and the semantics of that data, then it will be easier to evaluate the different proposed OODBMS data models in terms of their suitability for different applications. Then as standards are introduced to the community an exchange format will allow for the sharing of data in the common format, while still retaining the unique abilities of the various systems developed for specialized applications.

At Rensselaer, a tool has been developed [Spo] that can import certain data formats into the ROSE system. Then, using a data translation tool, the structure of the objects in the resulting ROSE database is altered to conform to the structure required for a target applications. Several data formats including FORTRAN Namelist and IGES files can be imported into ROSE, and then altered to work with existing ROSE applications. This work has demonstrated the need for a flexible means of data exchange, and has also high-lighted the significant information loss that can occur in such a translation.

The issue of semantic data loss will be the crucial issue in the translation of data between OODBMSs. It is the encapsulation of this semantic information, and the abstraction possible with object-oriented techniques that makes these models attractive. Any system which permits such information loss during data transfer will clearly not be acceptable. Thus both of the traditional methods of data exchange, common file formats and query translation, will fail to fully solve this problem. A related issue is exchange of data between object-oriented and non-object-oriented systems. The experiments we have conducted seem to show that the traditional file based exchange systems will work here.

I propose two approaches for data exchange in OODBMSs. The first approach deals with the exchange of data to and from a non-object-oriented system. Thus all that we are concerned with is the exchange of the data. This could be accomplished in a verbose format somewhat similar to the Namelist format from FORTRAN.

All hierarchical objects will be "flattened" first, and the various data items will be stored in the format:

<u>Attribute</u>: <u>value</u>

This would preserve the information from the naming schemes, and lead to a non-ambiguous transfer system. The format will be rather bulky, but with the compression tools available this should not present too much of an obstacle to data sharing.

The second format of exchange is an active one. The idea here being that the source and target DBMS are both active, and the information is exchanged via a Unix-like socket, with the data accessing routines in the source directly feeding the data definition methods in the target. The motivation for this approach is derived from the basic philosophy of object-oriented systems themselves, principally that of data encapsulation. The prospect of one DBMS directly accessing another DBMS's internal storage format is in direct violation of the principles of data abstraction and encapsulation. Instead, the source system's data accessing functions are used to extract primitive data values which are then passed to the constructor methods in the target system. This not only works within the framework of data encapsulation, but it also allows all of the data integrity constraints that are programmed into the constructor functions to be utilized. This will assure that the data transferred into the system will conform to that system's standards. To accomplish this, it will be necessary to augment the OODBMS with methods to export and import that can be used by all objects in the DBMS. They would take other methods as their arguments, and would have

the ability to transfer primitive data types (integers, text, etc.) from one active system to another. The basic idea of this approach is analogous to that of query translation in a federated database system.

In conclusion I would say that the time has not yet come for a pervasive set of OODBMS standards. However, it is not too early for us to be considering the various support mechanisms that such standards will need when they do come into existence. One of the areas that seems ready for such consideration is the form in which data exchange between OODBMSs, and between OODBMSs and non-OODBMSs will take place.

This work was partially supported by the National Science Foundation, Grant number DMC-8803252, and by the Defense Advanced Research Projects Agency Defense Sciences Office, DARPA Initiative in Concurrent Engineering, Contract Number MDA972-88-C-0047. All Opinions expressed or implied are those of the authors.

REFERENCES

Hardwick, M., D. Spooner, E. Hvannberg, B. Downie, A. Faulstich, D. Loffredo, A. Mehta, and D. Sanderson, "ROSE A Database System for Concurrent Engineering Applications," Proceedings of the Second Conference on Concurrent Engineering, CERC, West Virginia University, February 1990.

Spooner, D., D. Sanderson, and G. Charalambous, "A Data translation Tool for Engineering Systems," Proceedings 2nd International Conference on Data and Knowledge Systems for Manufacturing and Engineering, IEEE Computer Society Press, October 1989.

Wilkinson, K., P. Lyngbaek, and W. Hasan, "The Iris Architecture and Implemntation," IEEE Transactions on Knowledge and Data Engineering, vol. 2, pp. 63-75, IEEE Computer Society Press, March 1990.

Bretl, R., D. Maier, A. Otis, J. Penney , B. Schuchardt, J. Stein, E. Williams, and M. Williams, "The GemStone Data Managment System," in Object Oriented Concepts, Databases and Applications, ed. W. Kim & H. Lochovsky, pp. 283-308, ACM Press, New York, 1989.

NIST-114A (REV. 3-90)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 4503

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
FEBRUARY 1991

4. TITLE AND SUBTITLE

Proceedings of the Object-Oriented Database Task Group Workshop
Tuesday, May 22, 1990; Atlantic City, NJ

5. AUTHOR(S)
Elizabeth N. Fong, Editor
Craig W. Thompson, Editor (Texas Instruments Incorporated)

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)
U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)
National Institute of Standards and Technology
Gaithersburg, MD 20899

10. SUPPLEMENTARY NOTES

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

This report constitutes the proceedings of a one-day workshop on standardization of object database systems held in Atlantic City, New Jersey, on May 22, 1990. The workshop was sponsored by the Object-Oriented Database Task Group (OODBTG) of the ASC/X3/SPARC Database Systems Study Group (DBSSG).

This workshop, held the day before the ACM International Conference on Management of Data (SIGMOD '90 conference), was the first of two workshops held to solicit public input to identify what aspects of object database systems may be candidates for consensus that can lead to standards. The second companion workshop was held on October 23, 1990, in Ottawa, Canada, coincident with the Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA).

The workshop attempted to focus on concrete proposals for language or module interfaces, exchange mechanisms, abstract specifications, common libraries, or benchmarks. The workshop announcement also solicited papers on the relationship of object database system capabilities to existing standards, including assertions that question the wisdom of standardization.

This proceedings consists of 22 position papers covering various aspects where standardization on object database systems may be possible.

NIST is publishing the proceedings of both of these workshops to disseminate information on object standardization activities. The proceedings of the second workshop on standardization of object database systems appeared as NISTIR 4488.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)
database; database management system; DBMS; data model; object-oriented; OODB; programming languages; standards.

13. AVAILABILITY

- [X] UNLIMITED
- [ ] FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).
- [ ] ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.
- [X] ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
308

15. PRICE
A14

ELECTRONIC FORM