NAT'L INST. OF STAND & TECH
A11106 978007

COMPUTER SCIENCE & TECHNOLOGY:

National Bureau of Standards
Library, E-01 Admin. Bldg.
OCT 1 1981
191060
QC
100
.457

DATA ABSTRACTION, DATABASES, AND CONCEPTUAL MODELLING:

AN ANNOTATED BIBLIOGRAPHY

500-59
D

NBS Special Publication 500–59
U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards

NATIONAL BUREAU OF STANDARDS

The National Bureau of Standards[1] was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the Nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the Nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, and the Institute for Computer Sciences and Technology.

THE NATIONAL MEASUREMENT LABORATORY provides the national system of physical and chemical and materials measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; conducts materials research leading to improved methods of measurement, standards, and data on the properties of materials needed by industry, commerce, educational institutions, and Government; provides advisory and research services to other Government agencies; develops, produces, and distributes Standard Reference Materials; and provides calibration services. The Laboratory consists of the following centers:

Absolute Physical Quantities[2] — Radiation Research — Thermodynamics and Molecular Science — Analytical Chemistry — Materials Science.

THE NATIONAL ENGINEERING LABORATORY provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

Applied Mathematics — Electronics and Electrical Engineering[2] — Mechanical Engineering and Process Technology[2] — Building Technology — Fire Research — Consumer Product Technology — Field Methods.

THE INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following centers:

Programming Science and Technology — Computer Systems Engineering.

[1]Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Washington, DC 20234.
[2]Some divisions within the center are located at Boulder, CO 80303.

NATIONAL BUREAU
OF STANDARDS
LIBRARY
JUN 15 1981

COMPUTER SCIENCE & TECHNOLOGY:

DATA ABSTRACTION, DATABASES, AND CONCEPTUAL MODELLING:

AN ANNOTATED BIBLIOGRAPHY

Michael L. Brodie

Department of Computer Science
University of Maryland
College Park, MD 20740

NBS Special Publication

U.S. DEPARTMENT OF COMMERCE, Philip M. Klutznick, Secretary

Luther H. Hodges, Jr., Deputy Secretary

Jordan J. Baruch, Assistant Secretary for Productivity, Technology, and Innovation

NATIONAL BUREAU OF STANDARDS, Ernest Ambler, Director

Issued May 1980

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

National Bureau of Standards Special Publication 500–59
Nat. Bur. Stand. (U.S.), Spec. Publ. 500–59, 86 pages (May 1980)
CODEN: XNBSAV

Library of Congress Catalog Card Number: 80-600052

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1980

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
Price $3.75
(Add 25 percent additional for other than U.S. mailing)

PREFACE

Abstraction, one of the most common and most important intellectual activities, enables people to model and deal with the world around them. This observation is basic to modelling parts of the "real world" in computers. In light of this the topic of abstraction and its representation has been actively and fruitfully investigated by researchers in computer and information sciences. In particular, the three fields of artificial intelligence, databases, and programming languages have addressed overlapping issues within the area of conceptual models for dynamic systems of complex data. For example, in artificial intelligence there is knowledge representation, in databases there are semantic database models, and in programming languages there is data abstraction. Each group has made important contributions and has discovered problems in applying the work of other groups to their area. All groups are addressing aspects of the following open problems:

What should be modelled?
- What world (real or man-made) is being modelled?
- Which aspects are relevant to model?

How should what is being modelled be described?
- operationally or declaratively?
- with which semantic primitives?
- What is the role of 'types'?
- How is the structure of a system modelled?
- How does process modelling affect data modelling?
- How is the consistency and integrity of the model assured?

How are the relationships among/within models described?
- What kinds of relationships are there?
- How is consistency of model descriptions (e.g., specification with implementation) assured?

Issues in applying modelling techniques:
- embedding in or access from programming languages,
- adapting to the evolution of the system being modelled,
- supporting methodologies and tools,
- ability to interface with unlike systems.

This bibliography is intended to be a comprehensive reference to work on conceptual modelling of dynamic systems of complex data. A second objective is to encourage the cross-fertilization of the three research areas. Although each area has its own, unique concerns, there are important concepts, goals, and problems common to all three areas. The bibliography emphasizes one of these concepts, ***data abstraction***. It is hoped that this bibliography will encourage an exchange of information on the technologies being used to specify and represent conceptual models in each of the areas and on problems that are still to be solved.

The bibliogaphy contains entries for books, articles, and papers from the areas of programming languages, database management, artificial intelligence, and software engineering. Entries reference works that present:

- terminology, basic concepts, and open research problems in each area;

- approaches, in each area, to data modelling: concepts, tools, techniques, and open problems;

- the integration of approaches from two or more areas to data and its treatment through programming languages.

There is some emphasis on specification, representation, and verification issues that arise in the design, development, and maintenance of database applications.

The bibliography consists of three sections. The first is a list of the topic areas covered by referenced works. The second is an index, by topic, to the bibliographic entries. Entries are indexed by the first author's last name followed by the year and, possibly, a qualifying letter, e.g., **Paolini79a**. A citation of the form "In XXX" means that the work was published in a volume also included in the bibliography with index XXX, e.g., In Utah76. The third section contains the bibliographic entries. Some entries are annotated and some include the author's abstracts.

DATA ABSTRACTION, DATABASES, AND CONCEPTUAL MODELLING:

AN ANNOTATED BIBLIOGRAPHY

MICHAEL L. BRODIE

ABSTRACT

This bibliography contains entries for over 350 books, articles, and papers on issues within the area of conceptual modelling of dynamic systems of complex data. The entries have been drawn from recent work in the areas of database management, programming languages, artificial intelligence, and software engineering. The bibliography has two puposes: to present a comprehensive list of annotated references to research into issues of data abstraction, databases, and conceptual modelling; and second, to encourage the cross-fertilization of the three research areas of database management, programming languages, and artificial intelligence.

Key words: Abstract data types; artificial intelligence; data abstraction; database management systems; data structures; programming languages; software engineering.

ACKNOWLEDGEMENTS

The author is grateful for contributions and comments from Robert Balzer, Janis Bubenko Jr., John Gannon, and Lawrence Rowe. Special thanks are due to John Mylopoulos who compiled most of the AI references and to Nancy Sevitski for her assistance in text editing.

The author is also grateful to Donald R. Deutsch, Dennis Fife, and Edgar H. Sibley for their support in pursuing this research.

This work was supported, in part, by the National Bureau of Standards under contract number NBS79SBCA0216 and by the National Science Foundation under grant number MCS 77-22509. The Computer Science Center of the University of Maryland supported, in part, the production of this document.

TOPIC AREAS

Programming Languages

- general
- data abstraction
- specifications
- verification and validation
- programming methodology
- programming language design considerations

Database Management

- general
- database models
- database languages (definition, manipulation, query)
- logical database design
- specifications
- verification
- database architecture (multiple languages & schemas)
- formalization of database concepts
- concurrency and sharing

Artificial Intelligence

- representation of knowledge:
  - general
  - logical representations
  - network representations
  - procedural representations
- Artificial Intelligence programming languages

Software Engineering

- general
- application of software engineering to databases
- systems analysis and design

Relationship Between Programming Languages, Database Management, and Artificial Intelligence

- general
- data type concepts
- high level programming language access to databases
- application of data abstraction to databases
- applications of artificial intelligence to databases

Concepts of Data

- general
- advanced type concepts
- abstract data types
- data reliability
- stored definitions
- structural and procedural descriptions of systems
- modularity

TOPIC INDEX TO BIBLIOGRAPHIC ENTRIES

PROGRAMMING LANGUAGES

general:
Brinch75a; Leavenworth74; PSCC79; Wirth71a,75; Yeh77a;

data abstraction:
Ambler77; Atkinson78c; DeRemer79; Elliott78; Geschke78; Goldberg76; Good78; Guttag; Holt78,79; Horning76; Ingalls78; Johnson76; Lampson77,79; Ledgard77; Liskov77a,b,78; London78b; Mitchell79; Morris79; REDL78; Reynolds75; Shaw76,77; Shoch79; Standish77; Wasserman78b; Wells76; Wirth77a,b,c; Wulf76b;

specifications:
Ambler77; Ardis79; Balzer76,78,79a,b; Bartussek77; Berzins79; Buckle77; Burstall77; Ehrich78a,b; Ehrig77a,b; Goguen77,79b;Guttag; Linden78; Liskov75; Majster77,79a,79b; Mann79; Melliar79; Parnas70,77; Ram78a,b; Riddle78; Roubine76; Shaw77; Standish77; Thatcher78,79; Wortman79; Yeh77a; Zilles75;

verification and validation:
Bjorner78; Burstall72; DeMillo79; Flon76; Gerhart78; Goguen79a; Guttag; Hantler76; Hoare72b,73a,74; King79; London78a; Model79; Robinson77; Shaw77; Spitzen75,78; Wegbreit76; Yeh77a;

programming methodology:
Balzer67,73,76; Dahl72; DeRemer75; Dijkstra72; Goas78; Leavenworth79a,b; Linden76; Liskov72,74; MacEwen78; Manna79; Mealy77; Parnas70,72,75,79; Riddle78; Towster79; Wasserman79a,b; Wirth71b,74;

programming language design considerations:
Balzer79; Denvir79; Dijkstra78a,b; DOD78; Gannon75; Goldsmith74; GREEN78; Popek77; PLDRS77; Stonebraker77; Winograd79; Wirth75;

DATABASE MANAGEMENT

general:
British77; Chen79; Fry76; Guide70; Langefors77; Mealy67; SIGMOD75,76,77,78,79; Tsichritzis77; Ullman80; VLDB75,77,78,79;

database models:
Abrial74; Astrahan76; Bachman77,78a,b; Biller78; Brodie78a,79a; Bubenko79a; Chen76,79; Codd79; dosSantos79; Hammer78a; Kent78,79; Lind79; McGee76; McLeod78a; Mylopoulos78; Schmid75; Smith77a,78b,79; Solvberg79; Vassiliou79; Wong77;

database languages (definition, manipulation, query):
Buneman79; Chamberlin76; Codd71; Date76; Hammer77; McLeod76,78b; Merrett77; Mylopoulos78,79; Pirotte78; Prenner77,78; Rowe79; Schmidt77; Shipman80; Shopiro78; Shneiderman78; Stonebraker75,77; Wasserman78b,79c;

logical database design:
Berild77; Bubenko77a,79b; Chen79; Gerritsen78; IBM; Lind79; NYU78; Palmer78; Rolland79; Scheuermann79; Smith78c; Sungren74,78; Weber78; Wiederhold77; Wong79; Yeh77b,78a,b;

specifications:
Abrial74; Bjorner78b; Baldissera79; Balzer79b; Beeri78,79; Bernstein76; Brodie78a; Goldman79; Guttag76,77a,78; McLeod76; Paolini79b; Smith77a,b,78b,79; Weber78; Solvberg79;

verification:
Badal79; Baldissera79; Brodie78a; Bubenko77b; Guttag79b; Hammer78b; Maier79; Minker79;

database architecture (multiple languages & schemas):
Date76; Hammer79; Katz79; Paolini79a; Tsichritzis78;

formalization of database concepts:
Abrial74; Beeri78,79; Bernstein76; Bjorner78; Brodie78a,79a; Vassiliou79;

concurrency and sharing:
Dayal79; Schmidt79; Stonebraker75;

ARTIFICIAL INTELLIGENCE

representation of knowledge: general:
Bobrow75,77; Findler79; Hayes74; Irwin75; Minsky75; Roberts77; Szolovits77; Waterman79; Winograd74,75; Woods75;

representations of knowledge: logical representations:
Gallaire78; Kowalski74,76; Reiter76; Sandewall70; Schubert76;

representation of knowledge: network representations:
Brachman79; Fahlman79; Hendrix75; Levensque79; Norman75; Quillian68; Rieger76; Schubert76; Shapiro79; Simmons73;

representation of knowledge: procedural representations:
Davis75; Hewitt72,73; Winograd74,75;

artificial intelligence programming languages:
Balzer73; Bobrow74,77; Feldman72; Hewitt72; Model79; Shoch79; Sussman72; Wielinga78;

SOFTWARE ENGINEERING

general:
Parnas; Zelkowitz78,79;

application of software engineering to databases:
Brodie79b,c; Wasserman78a,79a; Weber78; Yeh77b,78a,b;

systems analysis and design:
Couger73; Gane79; Hoare72c; Jackson75; Jones79;; Ramamoorthy78a,b; Randell78; Ross77; Smith78a,79; Taggart77; Towster79; Warnier74; Wasserman79a; Yourdon79;

RELATIONSHIP BETWEEN PROGRAMMING LANGUAGES, DATABASE MANAGEMENT, and ARTIFICIAL INTELLIGENCE

general:
Brodie79b,c; dosSantos79;

data type concepts:
Brodie78a,b,c; McLeod76,78a; Rowe79; Schmidt77,78; Wasserman79d; Wulf76a;

high level programming language access to databases:
Allman76; Atkinson78b; Balzer79a; Bratsbergsengen77; Brodie78a,b; Buneman79; Hammer77; McLeod78b; Mylopoulos78; Prenner78; Rowe79; Schmidt77; Shopiro78; Wasserman78b;

application of data abstraction to databases:
Atkinson78a; Brodie78a,d,79b,c; Ehrig78; Hammer76a; Ledgard77; Lockemann79a,b; Manola77; Melkanoff78; Paolini79; Rowe79; Scheuermann79; Schmidt78; Wasserman78a,b,79d; Weber76,78;

applications of artificial intelligence to databases:
Gallaire78; Minker75; Mylopoulos79; Rossopoulos75,76; Wong77;

CONCEPTS of DATA

general:
Earley71; Hoare72a; Morris73; Nordstrom78; Ross76; Yeh77a;

advanced type concepts:
Brinch75b; Chang78; Demers78; Feldman78; Gries77; Hoare73b; Jones78; Mitchell76,77; Parnas76; Shaw76; Spitzen77; Wegbreit75; Wells76;

abstract data types:
Ardis79; Brand78; Brodie78d; Berzins79; Ehrich78a,b; Ehrig77a,b; Flon76; Geschke75; Gougen77,78; Guttag75,77b,78,79a,b; Linden76; Majster79b; Shaw76,78; Standish77; Thatcher79; Wasserman78a,79d; Weber78;

data reliability:
Brodie79d; Cousot77; Fosdick75; Goguen78; Goodenough75; Hammer75,76b; Hoare75; Holt79; Melliar77; Model79; Morris79; Randell78; Stonebraker75; Verhofstad78;

stored definitions:
Hammer79; Liskov77b,78;

structural and procedural descriptions of systems:
Hayes77; Manola77; Riddle78; Rolland79; Winograd75;

modularity:
Baker79; LeVankiet78; Parnas72;

BIBLIOGRAPHIC ENTRIES

Abrial74
Abrial, J. R. Data semantics. In Klimbie, J. W. and Koffeman, K. L. (Eds.), *Database Management*, North-Holland, 1974, pp. 1-59.

The author discusses the semantics of databases and presents a functional data model for their description. The work draws on research in generalized DBMSes, relational databases, and artificial intelligence. In the data model, the description of an object is given by the connections it has with other objects. Each connection is a pair of possibly multivalued access functions, one for each direction along the connection. Hence, the data model is based on binary relations.

The data model is defined informally using examples. The author discusses missing values (nothing, unknown), naming, side effects, standard or generic functions, and the writing of algorithms and programs within the model. The data model is formally defined in terms of mathematics and the basic concepts of the model (the model is used to define itself). Finally, physical implementation considerations are given. This paper was perhaps the first extensive investigation of data semantics and the first description of the functional approach to data semantics.

Allman76
Allman, E., Stonebraker, M., and Held, G. Embedding a relational data sublanguage in a general purpose programming language. In Utah76.

Ambler77
Ambler, A. L., Good, D. I., Browne, C., Burger, W. F., Cohen, R. M., Hoch, C. G., and Wells, R. E. GYPSY: A language for specification and verification of verifiable programs. *SIGPLAN Notices 12*, 3(March 1977), pp. 1-10.

An introduction to the Gypsy programming and specification language is given. Gypsy is a high-level programming language with facilities for general programming and also for systems programming that is oriented toward communications processing. This includes facilities for concurrent processes and process synchronization. Gypsy also contains facilities for detecting and processing errors that are due to the actual running of the program in an imperfect environment. The specification facilities give a precise way of expressing the desired properties of the Gypsy programs. All of the features of Gypsy are fully verifiable, either by formal proof or by validation at run time. An overview of the language design and a detailed example program are given. (authors' abstract)

Ardis79
Ardis, M. A. and Hamlet, R. G. Structure of specifications and implementations of data abstractions. Computer Science TR-801, University of Maryland, Sept. 1979.

A data abstraction is a collection of sets together with a collection of functions. An intuitive abstraction is unconnected with formalism: the sets and functions are supposed to be known *ab initio*. Formal ideas enter when the abstraction is (i) *implemented*, a conventional program written to carry out the operations on actual data; and (ii) *specified*, a mathematical characterization given to precisely describe its sets and functions. The intuitive abstraction, an implementation, and a specification share a syntax that names the sets and functions, and gives the function domains and ranges (as set names). The central question for any particular example of syntax is whether the semantics of the three ideas correspond: does the collection of objects and operations a human being was thinking of behave in the way the implementation's data and procedures behave? Do the mathematical entities behave as imagined? The

questions can never be answered precisely, because the intuitive abstraction is imprecise. On the other hand, precise comparison of specification and implementation is possible.

This paper presents an algebraic comparison of specifications with implementations. It is shown that these abstractions always overlap, and have a common (lattice) structure that is valuable in understanding the modification of code or specification. However, in dealing with the precise entities subject to formal analysis, we must not lose sight of the intuition behind them. Therefore, our definitions are framed in terms of the intuitive abstraction a person attempted to specify or implement, and we refer the algebraic ideas to this standard whenever possible.

Section 1 presents the intuitive ideas of an abstraction, its implementation, and specification. The ideas are essentially those of Hoare and Guttag. Section 2 gives the common formalism to be used, the constant word algebra. In Sections 3 and 4, this is applied to ideas, and suggests that the precise connection can shed light on the imprecise one that is really of interest: the intuitive abstraction in a person's mind. (authors' abstract)

Astrahan76
Astrahan, M. M., et al. System R: Relational approach to database management. *ACM TODS 1*, 2 (June 1976).

Atkinson78a
Atkinson, M., Martin, J. and Jordon, M. A uniform, modular structure for databases and programs. CSR-33-78, U. Edinburgh, Oct. 1978.

This paper discusses the relationship of a modular database description and the modular structure of programs over the database. It presents an approach to the design of large software systems that store data in a set of shared databases. The objective is to minimize the difference in treatment between data and programs. The authors investigate the relationship between abstract data types and database systems. The result is a modular structure in the data description which is refined in parallel with the program development. The approach is aimed at a top-down design of a database. The authors claim improvements for design quality, consistency, and completeness.

Atkinson78b
Atkinson, M. P. Programming languages and data bases. In VLDB78.

Research work in programming languages and database systems is combating the same problems of scale, change, and complexity. This paper looks at the present difficulties of relating persistent data with changing programs. It argues that there is a discontinuity between the data types and programming structures in existing programming languages and database systems. This discontinuity results in problems of programming methodology, translation, algorithm design, and implementation. The paper illustrates this discontinuity at the database interface by means of examples. The author proposes the need for new language primitives to encapsulate database concepts. Several such primitives are examined. It is suggested that these primitives could simplify the use of databases by programmers. ALGOL68 is used as a base language which is extended to accommodate high level access to databases via the mode control. The author proposes that all data be treated alike but that some data can be declared as persistent. He proposes that research in this direction should look for the simplest set of primitives to achieve the integration of databases and programming languages.

Atkinson78c
Atkinson, R. R., Liskov, B. H. and Scheifler, R. W. Aspects of implementing CLU. *Proc. 1978 ACM Annual Conf.*, Washington, D. C., Dec. 1978.

CLU is a programming language/system that supports procedural, data, and control abstractions. This paper reviews linguistic mechanisms in CLU which support structured exception handling, iteration over abstract objects, and parameterized abstractions. The implementation of these features is also discussed.

In CLU, a routine (procedure or iterator) can be programmed to terminate normally by executing a *return* statement or terminate in an exceptional condition by executing a *signal* statement. The exception is handled by the calling routine which must provide a handler - a list of exceptions and code to handle them. If an exception has no appropriate action associated, a failure exception is raised.

Iterators, a type of control abstraction, permit iteration over a collection of data items without knowing how they are obtained. A *for* statement is used to generate items one at a time and maintain the current state of the collection while routines use the current item.

Procedures, iterators, and clusters can all be parameterized. This supports abstraction by permitting one module declaration to be used for a class of related abstractions. Parameters are limited to a few types, however, size parameters are not needed since CLU objects can change size dynamically. Although the module does not know what the actual parameters will be, information about operations on the actual type may have to be provided in a *where* clause.

Bachman77
Bachman, C. W. and Daya, M. The role concept in data models. In VLDB77.

The authors present a new data model which is a direct extension of the network model, but is claimed to be a complete conceptual data model. The basic premise of the model is that database applications are collections of entities that are characterized by their behavior. An entity is defined as having a certain structure (as represented in traditional record types) and may play one or more roles. For example, a person or a corporation may be entities while a stockholder and a customer are roles that those entities may assume. The authors introduce role-segments and operations over roles to extend the CODASYL system to support roles. It is argued that the role model supports meta entities that are not supported in other models and therefore the role model is a good basis for conceptual schemas.

The role model provides solutions to existing CODASYL problems: multiple member set types can be expressed using sharable member roles, alternate owner set types can be expressed using sharable owner roles, recursive and "sometimes" sets are expressible. The authors claim that the role model will: facilitate schema mapping and the movement of data throughout an information system; provide greater semantic (integrity) power; and be compatible with high level programming languages.

The main points of the paper (in terms of data abstraction) are: emphasis on behavioral properties, inclusion of behavior in the data model and schema; semantic integrity; improved application semantics; the concept of "sub-entity" or shared roles; and the need for integrating data models with high level programming languages.

Bachman78a
Bachman, C. W. and Daya, M. Additional comments on the role model. Honeywell Information Systems Inc., Billerica, Mass., May 1978.

Bachman78b
Bachman, C. W. and Daya, M. The database manipulation primitives of the role model. Honeywell Information Systems Inc., Billerica, Mass., Oct. 1978.

Badal79
Badal, D. Z. and Popek, G. J. Cost and performance analysis of semantic integrity validation methods. In SIGMOD79.

Baker79
Baker, A.L. and Zweben, S.H. The use of software science in evaluating modularity concepts. *IEEE Trans. Soft. Eng. SE-5*, 2 (March 1979).

Baldissera79
Baldissera, C., Ceri, S., Pelagatti, G., and Bracchi, G. Interactive specification and formal verification of user's views in database design. In VLDB79.

Among the different phases of the database design process, the phase of modelling user's views has a particular relevance. This paper describes an interactive methodology for designing the views starting from the elementary sentences that specify the requirements of the application. The methodology generates a canonical representation, and provides verification algorithms for detecting inconsistencies, redundancies and ambiguities, and for restructuring and optimizing the model. (authors' abstract).

Balzer67
Balzer, R.M. Dataless programming. *Proc. 1967 FJCC*, pp. 535-544.

Balzer73
Balzer, R.M. A global view of automatic programming. *Proc International Joint Conf. on AI*, 1973.

Balzer76
Balzer, R.M., Goldman, N. and Wile, D. On the transformational approach to programming. In PSE76.

This paper discusses various approaches to programming, it defines and highlights Transformational Implementation. It then examines the basic causes of software problems and their resolution with the Transformational Implementation approach. Finally, an example illustrating the approach is given. (authors' abstract)

Balzer78
Balzer, R.M., Goldman, N. and Wile, D. Informality in program specifications. *IEEE Trans. on Software Engineering SE-4*, 2(March 1978).

This paper is concerned with the need for computer-based tools which help designers formulate formal process-oriented specifications. It first determines some attributes of a suitable process-oriented specification language, then examines the reasons why specifications would still be difficult to write in such a language in the absence of formulation tools. The key to overcoming these difficulties seems to be the careful introduction of informality based on partial, rather than complete, descriptions and the use of a computer-based tool that uses context extensively to complete these descriptions during the process of constructing a well-formed specification. Some results obtained by a running prototype of such a

computer-based tool on a few informal example specifications are presented and, finally, some of the techniques used by this prototype system are discussed. (authors' abstract)

Balzer79a
Balzer, R. and Goldman, N. Principles of good software specification and their implications for specification language. In PSRS79.

This paper examines the uses of and criteria for software specifications, which are then used to develop design principles for "good" specifications. These principles, in turn, imply a number of requirements for specification languages that strongly constrain the set of adequate specification languages and identify the need for several capabilities that are novel in the programming language area. The constraints imply the need for an ultra-high-level language which combines the database concept of a global data model containing alternate viewpoints with the control structures of programming languages.

The eight design principles are: 1) separate functionality from implementation; 2) a process-oriented systems specification language; 3) a specification must encompass the system containing the software; 4) a specification must encompass the environment in which the system operates; 5) a system specification must be a cognitive model; 6) a specification must be operational; 7) a system specification must be tolerant of incompleteness and must be augmentable; 8) a specification must be localized and loosely coupled.

The resulting 17 implications for specification languages include: a global model; a high level relational database; uniform data specification; global database with inference; descriptive, historical, and future references; demons; logical aggregation; and the elimination of variables.

Balzer79b
Balzer,R. An implementation methodology for semantic data base models. In Chen79.

The Data Base community faces the same software crisis as the rest of the programming community as the gap between conceptual semantic data base models, such as Entity-Relationsip models, and the underlying physical representation of these data base models rapidly widens. This trend is expected to continue as the semantic models become increasingly abstract and as more sophisticated concrete data structures and search techniques are utilized.

Among the various approaches to resolving the software problem, one seems particularly relevant to the data base community. Its relevance arises from the fact that the language with which it deals includes semantic data models. This particular approach is based on a more general methodology for systematically transforming conceptual specifications into efficient implementations that are guaranteed to be valid and for easily maintaining these implementations.

This paper describes this general implementation methodology, its specific application to a specification language which spans semantic data models, an example of the implementation of a specification in this language, and the extension of the approach required for data base applications. (author's abstract)

Bartussek77
Bartussek, W. and Parnas, D. Using traces to write abstract specifications for software modules. UNC Report 77-012, University of North Carolina at Chapel Hill, Dec. 1977.

A specification for a software module is a statement of the requirements that the final programs must meet. In this paper we concentrate on that portion of the specification that describes the interface

between the module being specified and other programs (or persons) that will interact with that module. Because of the complexity of software products, it is advantageous to be able to evaluate the design of this interface without reference to any possible implementations. The first sections of this paper present a new approach to the writing of black box specifications, illustrate it on a number of small examples, and discuss checking the completeness of a specification. Section VIII is a case history of a module design. Although the module is a simple one, the early specifications (written using an earlier notation) contained design flaws that were not detected in spite of the involvement of several persons in a series of discussions about the module. These errors are easily recognized using the method introduced in this paper. (author's abstract)

Beeri78
Beeri, C., Bernstein, P. A., and Goodman, N. A sophisticate's introduction to database normalization theory. In VLDB78.

Beeri79
Beeri, C. and Bernstein, P. A. Computational problems related to the design of normal form relational schemas. *ACM TODS 4*, 1 (March 1979).

Berild77
Berild, S. and Nachmens, S. CS4: A tool for data base design by infological simulation. In VLDB77.

The authors discuss a database design tool, CS4, which has been developed to support infological simulation. Infological simulation involves the use of pilot-implementations of a system to be used to improve the (logical) structure of the system before it is actually implemented. Emphasis is placed on the design of the conceptual schema and not on the system as a whole.

CS4 supports: an associative database, a high level general purpose language, procedures, list handling, and a command language to handle text, procedure libraries and procedure tests and initialization.

CS4 has been enthusiastically accepted by database designers in a number of practical applications. It permits evolutionary design as well as the testing of design decisions. Although CS4 supports the design of a conceptual schema of entity types, designers are not forced or encouraged to provide abstract operations. Procedures may be used to design queries and transactions but access into generic operations is also permitted. Emphasis is on the structure and relationships in a schema not on its constraints and operations.

Bernstein76
Bernstein, P. A. Synthesizing third normal form relations from functional dependencies. *ACM TODS 1*, 4 (Dec. 1976).

Berzins79
Berzins, V.A. Abstract model specifications for data abstractions. Ph.D. diss. MIT/LCS/TR-221, Massachusetts Institute of Technology, July 1979.

A data abstraction introduces a data type with a hidden representation. Specifications of data abstractions are required to allow the data to be described and used without reference to the underlying representation. There are two main approaches to specifying data abstractions, the abstract model approach and the axiomatic approach.

This thesis is concerned with the problems of formalizing and extending the abstract model approach. A formally defined language for writing abstract model specifications is presented. The correctness of an implementation with respect to an abstract model specification is defined, and a method for proving the correctness of implementations is proposed.

Our formulation treats data abstractions with operations that can dynamically create new data objects, modify the properties of existing data objects, and raise exception conditions when presented with unusual input values. (author's abstract)

Biller78
Biller, H. and Neuhold, E. J. Semantics of data bases: The semantics of data models. *Information Systems 3*, 1, 1978.

Bjorner78a
Bjorner, D. *Formalization of Data Base Models.* TR ID811, DCS, Tech. U. of Denmark, Dec. 1978.

The three leading data base models are examined: the relational, the hierarchical and the network data base models. Each is formally defined: first the underlying data model; then typical data definition, data manipulation and query languages. The paper begins with an introduction to the abstract notation language used by applying it to simple file systems. The paper ends by discussing realizations derived from the formal specifications. (author's abstract)

Bjorner78b
Bjorner, D. and Jones, C. B. *The Vienna Development Method: Meta Language.* Springer-Verlag, New York, 1978.

Bobrow74
Bobrow, D. G. and Collins, A. *Representation and Understanding.* Academic Press, 1975.

Bobrow75
Bobrow, D. G. and Raphael, B. New programming languages for Artificial Intelligence research. *ACM Computing Surveys 6*, 3 (Sept. 1974).

Bobrow77
Bobrow, D. G. and Winograd, T. An overview of KRL, a knowledge representation language, *Cognitive Science 1*, 1, 1977.

Brachman79
Brachman, R. On the epistemological status of semantic networks. In Findler79.

Brand78
Brand, D. A note on data abstractions. *SIGPLAN Notices 13*, 1 (Jan. 1978), pp. 21-24.

The author relates his experience using data abstractions in implementing a program verifier. He indicates difficulties with specifications, difficulties with programming (data abstractions were not compiler enforced) and some suggested remedies. He concludes that data type abstractions substantially increase the

programmer's confidence in his programs but that program size necessarily increases (approximately twofold).

Bratsbergsengen77
Bratsbergsengen, K. and Risnes, O. ASTRAL - A structured relational applications language. *Proc. SIMULA Users Conf.*, Sept. 1977.

Brinch75a
Brinch Hansen, P. The programming language Concurrent Pascal. *IEEE Trans. on Soft. Eng. SE-1*, 2 (June 1975).

Brinch75b
Brinch Hansen, P. Universal types in concurrent PASCAL. *Information Processing Letters 3*, 6 (July 1975), pp. 165-166.

British77
The British Computer Society Data Dictionary Systems Working Party Report. *SIGMOD RECORD 9*, 4 (Dec. 1977).

Brodie77
Brodie, M. L. and Tsichritzis, D. Database constraints. CSRG-78, University of Toronto, Feb. 1977.

The authors discuss the application of abstract data types to databases in terms of the DBMS problems of specifying, verifying, implementing, and supporting the evolution of coexisting user views. These problems are expressed in terms of abstract data types. The goal is to represent user views as a network of abstract objects accessible only through fixed abstract operations. An example of such a network is presented. The main advantage of this approach is seen as the increased ability to define, analyze, and enforce database constraints. The DBMS problems that were expressed in terms of abstract data types are expressed in terms of database constraints.

The DBMS problems pose the following requirements for the abstract data type approach: developing a set of generic data abstractions for conceptual data modelling; developing a methodology for decomposing database applications and constructing abstract data type networks; developing proof techniques for database abstractions (i.e., proof of consistency of structural and behavioral properties); methods of building abstractions from existing abstractions; methods of altering abstractions that support data independence.

The authors define database constraints in terms of predicate calculus as a basis of the specification and verification of database abstractions. The earlier problems are then expressed in terms of database constraints: obtain a complete set of generic constraints; parameterization of constraints; proof techniques for constraint satisfiability; transformation of constraint sets to construct consistent user views and proofs of view consistency.

Brodie78a
Brodie, M. L. Specification and verification of database semantic integrity. Ph.D. diss., CSRG-91, University of Toronto, March 1978.

Semantic integrity is fundamental to the correct application and use of database systems. Two conditions necessary for semantic integrity are: the logical database schema must be a consistent set of constraints and the database values must satisfy the constraints in the schema. Unfortunately, the techniques currently available for ensuring semantic integrity are inadequate. Constraint specification techniques are incomplete and *ad hoc*. Due to the absence of a uniform view of constraints and semantic integrity, databases are difficult to specify. Few, if any, techniques exist to verify that constraints are consistent or that databases satisfy the appropriate constraints. Consequently, the application of database technology is severely limited.

The author discusses the application of techniques from programming languages and artificial intelligence to the specification and verification of logical, non-behavioral properties of databases. Abstract data types, specification techniques, abstraction, and data modelling are extended for the particular problems of databases, notably, data independence, data sharing, problems of scale, and data semantics. The notion of a data type algebra is developed as a basis for a new data model. The data model and a corresponding specification language are presented and axiomatized.

A major difficulty in ensuring the semantic integrity of a database lies in ascertaining whether or not a specification is consistent. The axiomatically defined specification language acts as a uniform basis for a verification technique which extends well known techniques from programming languages.

Also discussed are the semantics of data, data modelling, and a database design methodology based on data abstraction and data modelling concepts from structured programming and artificial intelligence.

Brodie78b
Brodie, M. L. Data types and databases. IFSM TR #37, University of Maryland, Dec. 1978.

Database and data type concepts are extended mutually to improve the semantic capabilities of both data models and data type systems and to resolve apparent discrepencies between databases and programming languages. To meet database needs, data structuring is developed to form an algebra of data types. A semantically rich data model is introduced to show that data models can be expressed in terms of data types. Finally, a schema specification language is presented to demonstrate the power of data type tools for the definition of database schemas and for the maintenance of database semantic integrity. (author's abstract)

Brodie78c
Brodie, M. L. The application of data types to databases. Bericht Nr. 51, Fach. Informatik, Universitat Hamburg, Dec. 1978.

Many problems faced by database designers, users, and implementors are due, in part, to the informal database concepts which have developed pragmatically, somewhat independent of other areas of computer science. In particular, database semantics have been poorly understood. This paper applies programming language and artificial intelligence concepts to databases. Data type concepts are extended to accommodate databases and *vice versa*. The result is a semantically rich data model, based on data type concepts, and a schema specification language which integrates these concepts. This approach permits data type concepts to be applied directly to databases. The role of axiomatization to formalize both the data model and its data language is described. Software engineering tools are applied to database design.

Brodie78d
Brodie, M. L. and Schmidt, J. W. What is the use of abstract data types? In VLDB78.

The authors discuss the application of abstract data types to large, shared, integrated databases. Abstract data type concepts seem closely related to the database issues of data independence, complete definition of semantics, and database architecture and implementaton problems. Although the use of fixed operations seems consistent with conceptual schema and semantic integrity goals, large problems are raised due to the complexity of database applications.

The authors argue that data abstraction concepts may provide direct benefits for database design methodologies, database semantics, and user interfaces. They provide one of the few approaches to the specification and verification of database application properties. Database architecture, data sharing, and concurrency problems are noted as areas of research for which abstract data type concepts must be extended.

Brodie79a
Brodie, M. L. Axiomatic definitions of data model semantics. IFSM TR #41, University of Maryland, Feb. 1979.

The axiomatic method, a widely accepted technique for the formal definition of programming language semantics, is used to define data model semantics. First, a definition of the term "data model" is developed. The strong relationship between database and programming language concepts is discussed. The nature of data model formalization is described. Based on the experience in programming languages, it is argued that the formal definition of a data model will provide important benefits for (i) database design, (ii) database management system implementation, (iii) semantic integrity verification and validation, and (iv) data model theory. Several different formal description techniques and their particular advantages are mentioned. It is argued that in order to achieve desired goals more than one technique be used to develop consistent and complementary formal definitions of a data model. The axiomatic method is described. Axiomatic definitions are particularly appropriate for the design, analysis, and comparison of schemas, programs, and databases. They provide advantages for implementation and data model theory. The axiomatic definition technique is demonstrated in a formal definition of the semantics of the structural aspects of a non-trivial data model which is based on the relational data model.

Brodie79b
Brodie, M. L. The Application of Programming Language and Software Engineering Tools and Techniques to the Development of Reliable Information Systems. IFSM Dept., University of Maryland, May 1979.

Despite the considerable advances achieved over the past two decades in the area of database management there is strong evidence that the traditional concepts for and approach to database problems are severely limited.

A major aspect of the traditional approach to database management is the emphasis on structural properties of database applications which has been almost exclusive of concern for the behavioral properties. This paper addresses the following four weaknesses of the approach. They are:
1. the inability to specify and verify database semantic integrity;
2. the inability to deal with behavioral properties of databases;
3. the lack of a comprehensive framework for the development of database applications; and
4. problems of database programming.

The paper proposes design tools and techniques for the development of reliable information systems which are based on a database management system. Reliability, or database semantic integrity, is the fundamental requirement of a database system and is the main goal to be achieved with the tools and techniques. Secondary goals relate to automatic programming, data independence, and efficiency.

The development of database concepts has been substantially independent of the areas of programming languages and software engineering. However, these two areas contain a large body of knowledge and experience with problems closely related to the above database problems. Research is required to consider the applicability of tools and techniques from programming languages and software engineering to the development of database applications. In particular, the concept of data abstraction should be used to integrate both structural and behavioral properties of databases.

Brodie79c
Brodie, M.L. A Research Environment for Investigating the Relationship between Database Management Systems and Programming Languages. IFSM Dept., University of Maryland, May 1979.

Database management systems and programming languages present many similar problems and goals; however, the two research areas have had quite separate developments. Over the past few years, there has been an increased awareness of the strong relationship between the two areas. As a result there have been some significant contributions on issues in the intersection of the two areas. It is now clear that database concepts and requirements will have a significant impact on programming language concepts and *vice versa*. Unfortunately, current standards activities in both areas have completely ignored this potential impact. Therefore, it is essential to understand the relationship between the two areas.

It is argued that a particular research environment is necessary for this research. Such an environment requires a conceptual framework and the establishment of problems, concepts, goals, and research directions. Secondly, it is necessary to have software tools with which to gain practical experience and to develop and test new tools and techniques.

Brodie79d
Brodie, M. L. Data Quality: Data Reliability and Semantic Integrity. IFSM TR# 42, University of Maryland, June 1979.

Reliability and integrity are fundamental to the correct development and use of software systems. This principle has been the major motivating force in the area of software engineering. Although the problems of reliability, integrity, and correctness are difficult, major advances have been made in terms of tools and techniques to increase reliability. Most of these advances have been related with programming languages and in particular with the behavioral aspects of software.

Database management systems were developed, somewhat independently of programming languages, to address the problems of data as the central issue. Within the database area, tools and techniques were developed to aid in the design, development, and use of data-oriented systems. Although there are many tools and techniques for data quality in databases, they are still at an early stage of use and development. "Perhaps the most neglected objective of DBMS is the maintenance of (data) quality [24]." Currently, there is considerable research to improve these tools and techniques.

The paper discusses the nature and importance of data quality. The role of data in the development of reliable software is discussed in terms of the software development life cycle. Existing tools and techniques for data quality from programming languages and database systems are surveyed. Limitations of current facilities and subsequent research directions are outlined for each step of the software development life cycle. Finally, it is argued that neither the database emphasis on structure (i.e., data) nor the software engineering emphasis on behavior is adequate, on its own, to achieve data quality in databases.

Bubenko77a
Bubenko, J. A. IAM: An inferential abstract modelling approach to design of conceptual schema. In SIGMOD77.

Bubenko77b
Bubenko, J. A. Validity and verification aspects of information modelling. In VLDB77.

Bubenko79a
Bubenko, J. Data models and their semantics. INFOTECH STATE-OF- THE-ART Report on data design, Sept. 1979.

Bubenko79b
Bubenko, J. A. On the role of "understanding models" in conceptual schema design. In VLDB79.

This paper suggests that two levels should be considered when designing a conceptual schema: the "understanding level" and the "conceptual database level." The first level includes two realms of importance

- the set of information requirements (IRQ) and
- the conceptual *information* model (CIM).

The next level includes two realms:

- the conceptual *database* model (CDBM) and
- the conceptual *processing* model (CPM).

The CIM is a full time-perspective, unrestricted view of the enterprise and aims at defining relevant relationships, events and inference relations. The CDBM is, together with CPM, a database realization of CIM which satisfies IRQ. In this paper, designs at both levels are illustrated and the utility of the conceptual information model is discussed. (author's abstract)

Buckle77
Buckle, N. Restricted data types, specification and enforcement of invariant properties of variables. In PLDRS77.

Buneman79
Buneman, P. and Frankel, R.E. FQL - A functional query language. In SIGMOD79.

This paper describes a purely applicative approach to database queries, based on the functional programming concepts of John Backus. The approach is realized in the language FQL which is currently implemented as an interface to a CODASYL system. The authors claim that FQL provides a sufficiently general data model to be an interface to other DBMSes. The main advantage claimed for the approach is that it provides a powerful formalism for expressing even very complex database queries.

FQL has no explicit references to data and provides a "functional schema". Information about any objects in the database (attribute values, relationships, etc.) is returned as the result of functions. The only control structure is the ability to combine functions. This approach differs from conventional query languages as follows: no notion of data currency, incremental development of complex queries, full computational power, and independence of the language and the DBMS. The authors claim an ability to encorporate semantic data models such as extensions of the relational model but acknowledge difficulties in supporting update.

Burstall72
Burstall, R. M. Some techniques for proving program correctness of programs which alter data structure. *Machine Intelligence 7*, (1972) pp. 23-50.

The paper extends Floyd's method to programs which alter data structures such as arrays, linear lists, or binary trees. The simple assignment axiom being insufficient, a family of distinct, specialized rules for assignment is introduced. These rules depend on the kind of data structure which is altered, and on the selection or construction used as the destination of the assignment; they are formulated in terms of a "change set" which defines identifiers whose meaning is changed by the assignment. A list-processing machine and its states are formalized in a first-order language; this formalization is applied to the proof of a flowchart program reversing linear lists. The result of the experiment being "long and tedious" and the proofs being "quite long and very boring," a better formalization is defined, based on a graph-like representation of the data structure. The assignment rules are reformulated in terms of the new formalization and the example is then proved cleanly and completely. The paper continues by generalizing the same methods for binary trees, as an example of more complex data structures, using the techniques of the category theory.

The proposed assignment rules are sensible extensions of the standard assignment axiom, but they are very ad hoc: how many such rules would be needed in the case of freely composed hierarchical data structures? One would prefer a more general assignment rule whose structure is not more complex than a unified method for defining various data structures. The various formalizations presented are essentially logical formulations of implementation-oriented representations. The author makes a valid point by showing how both the usefulness and the readability of a formalization depend on the elimination of unessential features. The use of the category theory may be an interesting exercise, as is usual with that theory, but here it brings no new results, just a simple generalization of the previous ones; moreover, its use for proving a tree-reversing program is inconclusive, because this program is recursive and is easily verified by an immediate application of recursion induction. (CR25, 707 M. Sintzoff)

Burstall77
Burstall, R. W. and Goguen, J. Putting theories together to make specifications. *Proc Fifth IJCAI*, Boston, Aug. 1977.

Chamberlin76
Chamberlin, D. DE., Astrahan, M. M., Eswaren, K. P., Griffiths, P. P., Lorie, R. A., Mehl, J. W. Reisner, P. and Wade, B. W. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM J. of Research and Development 20*, 6, Nov. 1976.

Chang78
Chang, E., Kaden, N. E. and Elliott, W. D. Abstract data types in EUCLID. *SIGPLAN Notices 13*, 3 (March 1978).

This paper assesses the data abstraction facilities of EUCLID. The authors claim that EUCLID supports abstract data types but does not, strictly speaking, support data abstraction. Five main points are made: (1) unlike CLU clusters, and ALPHARD forms, but like SIMULA 67 classes, EUCLID modules are not abstract data types since some representation details are not hidden. However, a programming discipline can be used to support abstract data types. (2) Unlike CLU, MESA, and ALPHARD, there is no enforceable separation between abstract definition (specification) and representation. The authors argue that the "ill-defined" abstraction function provides a means of specifying the relationship between the concrete representation and abstract definition. (3) Types in EUCLID can have formal parameters which must be typed constants; types cannot be passed as parameters to parameterized types. Hence, EUCLID does not support "schemes" or "polymorphic" types. (4) In EUCLID, operators can be added only as procedures;

operators (e.g., equality) cannot be redefined. (5) EUCLID reinforces the ALGOL-like block structure with further restrictions, including closed scopes (imports and exports lists), prohibits redeclaration of variables within scopes, and disallows side effects. It is argued that EUCLID struck a balance between providing full abstraction and simplifying implementation.

Chen76
Chen, P. P. S. The entity-relationship model: Towards a unified view of data, *ACM TODS 1*, 1 (March 1976).

Chen79
Chen, P.P. (Ed.) *Proc. Int'l Conf. on Entity-Relationship Approach to Systems Analysis and Design*, Los Angeles, Dec. 1979.

CODASYL
CODASYL Data Description Language Committee, Journal of Development 1978.

Codd71
Codd, E. F. Relational completeness of data base sublanguages. In Rustin, R.(Ed.) *Data Base Systems*, Prentice-Hall, 1971.

Codd79
Codd, E. F. Extending the database relational model to capture more meaning. *ACM TODS 4*, 4 (Dec.1979).

During the last three or four years, several investigators have been exploring "semantic models" for formatted databases. The intent is to capture (in a more or less formal way) more of the meaning of the data, so that database design can become more systematic and the database system itself can behave more intelligently. Two major thrusts are clear: (1) the search for meaningful units that are as small as possible -- *atomic semantics*; (2) the search for meaningful units that are larger than the usual n-ary relation -- *molecular semantics*.

In this paper, the author proposes extensions to the relational model to support certain atomic and molecular semantics. These extensions represent a synthesis of many ideas from the published work in semantic modelling plus the introduction of algebraic operators corresponding to a constrained second-order predicate logic. (author's abstract)

The work synthesized in this proposal includes: the dependencies and normal forms of Bernstein, Beeri, Fagin, et al.; Codd's null substitution principle; surrogates of Hall and Todd; the classification scheme of Schmidt and Swenson; aggregation and generalization of Smith and Smith; and cover aggregation of Hammer and McLeod.

Codd's main contribution here is to synthesize these primarily structural concepts and to provide operations necessary to complete the new data model. The author argues that the relational database model consists of a collection of time varying tabular relations (with given properties) and a relational algebra with which to define and manipulate relations. This paper introduces the notion that meta data as well as data is to be included in the data model. The paper is predicated on the need to provide users with high level abstractions defined in terms of structure and behavior.

Couger73
Couger, J. D. Evolution of business systems analysis. *ACM Computing Surveys 5*, 3 (Sept. 1973).

Cousot77
Cousot, P. and Cousot, R. Static determination of dynamic properties of generalized type unions. *SIGPLAN Notices 12*, 3 (March 1977).

Dahl72
Dahl, O. J., Dijkstra, E. W. and Hoare C. A. R. *APIC Studies in Data Processing No. 8: Structured Programming*, Academic Press, New York, 1972.

This book, a collection of three monographs, is one of the classic introductions to structured programming and concepts of abstraction.

The first monograph entitled "Notes on Structured Programming," by Dijkstra, addresses the principles and practices of structured programming that lead to improved quality and reliability of large programs. Although the discussions are somewhat informal, some topics are presented formally, e.g., programming examples and arguments about program proofs. Two examples of step-wise composition are presented.

The second monograph is discussed in Hoare72a.

In the third monograph, entitled "Hierarchical Program Structures," by Dahl and Hoare, the concepts of the first two monographs are related. It describes the merging of structured data and program control concepts within the class concept of SIMULA67. The class concepts are discussed and illustrated by examples. It is shown how classes can be used to encapsulate data structures and operations on them and how instances of such structures can be generated. The SIMULA class was one of the first mechanisms for data abstraction, however, unlike many of its successors, it permitted access to variables of a class object from outside the object.

Date76
Date, C. J. An architecture for high-level database extensions. In SIGMOD76.

This paper describes an achitecture for a set of database extensions to the existing high-level languages. The scheme described forms an achitecture in the sense that it is not based on any particular language: its constructs and functions, or some suitable subset of them, may be mapped into the concrete syntax of a number of distinct languages, such as COBOL and PL/1. The architecture includes both the means for specifying the programmer's view of a database (i.e. for defining the external schema) and the means for manipulating that view. A significant feature is that the programmer is provided with the ability to handle all three of the well-known database structures (relational, hierarchical, network), in a single integrated set of language extensions. Another important aspect is that both record- and set-level operations are provided, again in an integrated fashion. The objectives of the architecture are to show that it is possible for relational, hierarchical and network support to co-exist within a single language, and also, by providing a common framework and treating the three structures in a uniform manner, to shed some new light on the continuing debate on the relative merits of each. The paper is intended as an informal introduction to the architecture, and to this end includes several illustrative examples which make use of PL/1-based concrete syntax. (author's abstract)

The author provides several convincing arguments that high-level programming language users be able to access databases directly just as non-shared, non-persistent data is accessed. In this attempt at finding a common language to access hierarchic, relational, and network databases, the author drops a

number of features from DBTG and IMS, e. g., repeating groups, areas, realms, record ordering. These deletions result in a DDL and DML which are much simpler than those of DBTG and DML. Basically, the relational facilities are a subset of the hierarchic facilities which are a subset of the network language extensions.

Davis75
Davis, R. and King, J. An overview of production systems. AIM-271, Stanford University, AI Lab., 1975.

Dayal79
Dayal, U. and Bernstein, P. A. On the updatability of relational views. In SIGMOD79.

Demers78
Demers, A., Donahue, J., and Skinner, G. Data types as values: Polymorphism, type-checking, encapsulation. *Conf. Record 5th ACM Symp. on Principles of Programming Languages*, 1978.

The paper describes a new approach to the treatment of data types in programming languages which allows a simple interpretation of "polymorphic" or "generic" procedures, makes a simple set of type-checking rules semantically justifiable and provides a straight-forward treatment of encapsulation. (from authors' abstract)

The authors define a data type (following Scott's work) as "a set of operations specifying an interpretation of values of a universal value space." These data types are themselves elements of a universal domain. Variables are not considered. Data types are treated as arguments to procedures, functions, and data types.

The syntax and semantics of polymorphic procedures are given. Two important aspects are the *type* parameter which states that the corresponding argument must be a data type value and an associated *with* clause which lists the operations which must be provided with the type. The *with* clause permits static type-checking and specifies the transformation to be performed on its type argument when it is passed (i.e., a very simple coercion). This mechanism offers economies of concept (types treated as value parameters) and resources (one type check of all calls).

This approach to types permits a simple rule for type-compatibility based on the "principle of correspondence," i.e., declarative and parametric forms should be semantically equivalent. The authors' goal in terms of type-checking is representation-independence. Their approach is claimed to be more flexible than the macro-expansion view (as in EUCLID).

RUSSEL, the authors' language, supports encapsulation to permit the separation of a "concrete realization" from its use in a program. Like EUCLID and unlike ALPHARD and CLU, anything declared within a type can be hidden or exported and there is no explicit "representation" component. The emphasis throughout the paper is to consolidate to achieve an economy of concepts. In particular, capsules are explained in terms of types and polymorphism.

DeMillo79
De Millo, R. A., Lipton, R. J. and Perlis, A. J. Social Processes and proofs of theorems and programs. *Comm. ACM 22*, 5 (May 1979).

Denvir79
Denvir, B.T. On orthogonality in programming languages. *SIGPLAN Notices 14*, 7 (July 1979).

DeRemer75
DeRemer, F. and Kron, H. Programming-in-the-large versus programming-in-the-small. In PSE75.

The authors distinguish the activity of writing large programs from that of writing small ones. By large programs they mean systems consisting of many small programs (modules), possibly written by different people.

It is argued that languages for programming-in-the-small are needed, i.e. languages not unlike the common programming languages of today, for writing modules. There is a need for a "module interconnection language" for knitting those modules together into an integrated whole and for providing an overview that formally records the intent of the programer(s) and that can be checked for consistency by a compiler.

The authors explore the software reliablity aspects of such an interconnection language. Emphasis is placed on facilities for information hiding and for defining layers of virtual machines. (authors' abstract)

Many of the concepts addressed in this paper relate closely to problems that would be encountered in a modular approach to databases. In particular, module interconnections and user interfaces pose similar problems.

DeRemer79
DeRemer, F. and Levy, P. (Eds.) Summary of the characteristics of several "modern" programming languages. *SIGPLAN Notices 14*, 5 (1979), pp. 28-45.

This report is a brief survey and comparison of ALPHARD, ASPEN, CLU, EUCLID, GYPSY, MODULA, PASCAL, SIMULA 67, and SMALLTALK. Abstraction capabilities are noted in particular and each language is used to program a hash-coded string table.

Dijkstra72
Dijkstra, E. W. The humble programmer. *Comm. ACM 15*, 10 (Oct. 1972).

This 1972 Turing Award Lecture presents the author's philosophy of computer programming. It is proposed that in order to construct reliable software, programmers must confine themselves to intellectually managable tasks by means of "abstraction." It is argued that any problem can be factored into a hierarchy of (levels of) abstractions and that tools can be developed to create and integrate abstractions to construct larger programs. These arguments are based on the need and feasibility of designing, implementing, and increasing one's confidence in the correctness of computer programs. The author states "We must not forget that it is *not* our business to make programs; it is our business to design classes of computations that will display a desired behavior."

These principles have not yet, in 1979, been met for programming languages. The problems themselves, particularly concerning intellectual managability, are even more severe in the case of large database applications.

Dijkstra78a
Dijkstra, E. W. DoD-1: The summing up. *SIGPLAN Notices 13*, 7 (July 1978).

The author sums up his reaction to the DoD IRONMAN requirements and the four candidate languages. He makes the following points: first, the language designers were unwilling to criticize the requirements (e.g., it was not a requirement; it was a description of languages features; it was too specific; "strongly typed" was undefined; there was no firm position on side effects or aliasing), rather, they followed the requirements too closely. Second, IRONMAN mixed language design (goals) with compiler implementation; languages should be designed independently of implementation concerns. Third, languages based on PASCAL cannot improve on PASCAL until the notion of type is resolved ("The DoD would have gotten much more value for its money -- I guess -- if it had sponsored a few research contracts aimed at exploring whether the vague notion of 'strongly typed' can be given a clean, purposeful content, and if so, which are the options.") He applied this argument equally to aliasing, side effects, and enforceable language restrictions.

Dijkstra78b
Dijkstra, E. W. On the BLUE, GREEN, YELLOW, and RED (L) languages submitted to the DoD. *SIGPLAN Notices 13*, 10 (October 1978).

The author comments on each of the four languages submitted to DoD under the IRONMAN requirements.

BLUE follows the requirements which mistakenly mix language definition with compiler implementation. Specific criticisms focus on the complexity of BLUE and on its failing with regard to types and operators. Type compatibility is criticized for its matching algorithm and the definition of assignment compatibility of record variables. Problems with manifest expressions affect type identity and undo the advantages of parameterized types.

GREEN is criticized for its complexity, incompleteness, and lack of conciseness. Most criticisms relate to types, e.g., subtypes do not overcome PASCAL's problems; the notions of same type and identical type. Boxes (synchronization primitives) and generic program units are also seen as problematic.

YELLOW and RED are also criticized to a large degree based on their notion of type. Although RED claims to be strongly typed, type values (hence compatibility) can be determined only at run time. Also the notion of same type is unclear.

DoD78
Department of Defense Requirements for High Order Computer Programming Language, "Steelman," June 1978, U.S. Department of Defense, Washington, D.C.

dosSantos79
dos Santos, C. S., Neuhold, E. J., and Furtado, A. L. The data type approach to the entity-relationship model. *Proc. Int'l Conf. on E-R Approach to Systems Analysis and Design*, Los Angeles, Dec. 1979.

Earley71
Earley, J. Towards an understanding of data structures. *Comm. ACM 14*, 10 (Oct. 1971), pp. 617-627.

Ehrich78a
Ehrich, H. -D. Extensions and implementations of abstract data type specifications. *Notes in Computer Science 64*, Springer-Verlag, 1978.

Ehrich78b
Ehrich, H. -D. and Lohberger, V. G. Algebraic specification of databases and concurrent access synchronization, TR58/78, U. Dortmund, 1978.

Typical database concepts are specified algebraically by giving systems of equations for the basic update and retrieval operations. The authors attempt to show how database models can be constructed by putting such specifications together, combining them, and enriching them by integrity constraints. Integrity constraints are expressed directly in algebraic equations. Special attention is paid to the algebraic specification of concurrent access to shared resources and access restrictions due to privacy.

This is done by demonstrating the ability of algebraic specifications to express the behavior of locking, unlocking, and access restriction. The authors make no attempt to relate specifications to implementations but claim that their specifications can be extended to meet real needs and provide a basis for proving implementations correct. They also note the existence of problems with consistency and (sufficient) completeness of algebraic specifications.

Ehrig77a
Ehrig, H., Kreowski, H. J. and Padowitz, P. Some remarks concerning correct specification and implementation of abstract data types. TR77-13, Technical Univ. Berlin, 1977.

Ehrig77b
Ehrig, H. Stepwise specification and implementation of abstract data types. Technical Univ. Berlin, Feb. 1977.

Ehrig78
Ehrig, H. and Kreowski, H. J. and Weber, H. Algebraic specification schemas for data base systems. In VLDB78.

This paper proposes a hierarchical structuring principle for the formal specification of database systems, called an algebraic specification scheme, which is based on algebraic specification techniques for data types. Syntax and semantics of an algebraic specification scheme are formally defined and illustrated by a non-trivial example, the specification of a database system for an airport schedule. The construction of the schema is based on tuple- and table-connections of the components showing a close relationship to the relational database model. Algebraic specification schemes allow the definition of integrity constraints like functional and interrelational dependencies. The mathematically precise formulation allows rigorous correctness proofs for their syntax and semantics. (from authors' abstract)

The assumption that a database system can be designed top-down, in a hierarchical fashion has been questioned by several researchers (e.g., R. T. Yeh, J. M. Smith); however, a degree of bottom-up design is also included. The authors claim that mappings between conceptual and external levels can also be specified in using their technique.

Elliott78
Elliott, W. D. and Thompson, D. H. EUCLID and MODULA, *SIGPLAN Notices 13*, 3 (March 1978).

This paper discusses the design goals of both EUCLID and MODULA, two programming languages intended for writing software systems. Individual features of the languages are contrasted and a detailed comparison of modules and multiprogramming is given. MODULA differs from PASCAL in its modules (encapsulation mechanisms similar to EUCLID modules) and multiprogramming features which EUCLID does not offer.

EUCLID modules were designed primarily for data abstraction with the possibility of multiple instantiations, whereas MODULA modules were designed for encapsulation of entities that have only one instance. The MODULA module was intended to permit the establishment of static scope. EUCLID and MODULA modules have explicit imports and exports (use and define) lists with which to control the visibility of identifiers. However, MODULA limits access to variables and structural information more than does EUCLID. MODULA provides read-only access to and no field names or structural information for variables outside their modules. Other than modules and a slight variation in type compatibility (same type), the type concepts of the two languages are similar.

Fahlman79
Fahlman, S. *NETL: A system for representing and using real-world knowledge*, MIT Press, 1979.

Feldman72
Feldman, J. A., Low, J. R., Swinehart, D. C., and Taylor, R. H. Recent developments in SAIL. *Proc. AFIPS Fall Joint Conf.*, 1972, pp. 1193-1202.

Feldman78
Feldman, J. A. and Williams, G. J. Some remarks on data types. TR28, OCS, U. of Rochester, April 1978.

The authors address the following problem: current data type machinery does not permit the expression of constraints so that they can be checked automatically and efficient code can be produced, e.g., variables must have *one* type or be the same or coerced to the same type to match for operations. Their solution is to use explicit rather than implicit definition of properties of variables so that current optimization and verification capabilities can be used. Type definitions must include explicit statements of properties, e.g., small, odd, even, positive, legal coercions, procedures for complex coercions, and propagation rules. Syntax is proposed for such declarations as a basis for a language called ZENO.

Structure is to be defined in three levels: (i) bare skeleton level at which primitives exist; (ii) data structure level used to define the fixed or varying size, uniform or mixed constituents, and access type for the structure; and (iii) data type level to define all details to be fixed at compile time, including explicit definition of properties, assumptions, and assertions for the resulting type. The authors claim that current techniques can be used to check consistency of the defined properties (verification) and optimize generated code. The authors conclude with a list of outstanding problems in the development of ZENO: semantics of assertions, encapsulation, procedure typing and parameterization, and parallel processes.

Findler79
Findler, N. (Ed.) *Associative Networks*, Academic Press, 1979.

Flon76
Flon, L. and Haberman, A.N. Towards the construction of verifiable software systems. In Utah76.

Data types are an important design tool because they allow freedom of abstraction. Thus, they are useful for constructing large software systems, including operating systems. It is shown that when dealing with problems of concurrency, the use of path expressions, which are associated with data, makes the task of verification simpler than when the synchronization conditions are associated with programs. (authors' abstract)

The paper concerns the use of abstract data types in program verification. In particular, sequences of type operations called path expressions are used to address problems of concurrency in operating systems. The authors substantiate some of their claims by means of programmed examples.

Fosdick75
Fosdick, L. D. and Osterweil, L. J. Data flow analysis in software reliability. *ACM Computing Surveys 8*, 3 (Sept. 1975).

Fry76
Fry, J. P. and Sibley, E. H. Evolution of database management systems. *ACM Computing Surveys 8*, 1 (March 1976).

Gallaire78
Gallaire, H. and Minker, J. (Eds.) *Logic and Databases*, Plenum, 1972.

Gane79
Gane, C. and Sarson, T. *Structured Systems Analysis: Tools and techniques*, Prentice-Hall, Englewood Cliffs, N.J., 1979.

Gannon75
Gannon, J. D. and Horning, J. J. Language design for programming reliability. *IEEE Trans. on Soft. Eng. 1*, 2 (1975) pp. 179-191.

This well-written and readable paper presents a survey of sensible views about the relationship between programming language design and program reliability, and supports these views with a wide-ranging bibliography. The main part of the paper reports a well-conducted experiment comparing the performance of two groups of students in an operating systems course using two specially designed versions of a teaching language. The versions differed mainly in syntactic detail, but one version permitted more compile-time checks against semantic errors. The most significant results of the experiment confirmed the common sense belief that errors detected at compile time are removed more quickly, and the student learns faster not to make them, even though the crude number of errors may be higher at the beginning. The paper is also quite frank about the dangers and difficulties of such studies; that they depend on the student's previous conditioning, and on small samples from a highly nonhomogeneous population, highly subject to disturbance by extraneous factors. Further dangers are the extrapolation of results from students to more experienced, or even professional programmers, and the use of results derived from extremely high error rates to situations in which (one hopes) error rates are very close to zero.

The authors conclude that there remains much to be done in the design of languages for reliable software, and in the experimental evaluation of language design decisions, particularly to discover the effects of different environments. But it is important to remember that experiments must be designed to test

hypotheses and theories of language; and it is the theories that should dictate the design principles and the details of a programming language. An attempt to design a language by mere experimentation would be futile and ridiculous. The authors' eminently sensible theories have been supported by their experiments, and future experimentors can safely use their projects as a model. (CR29, 414 C. A. R. Hoare)

Gerhart78
Gerhart, S. L. Program verification in the 1980's: Problems, perspectives, and opportunities. ISI/RR-78-71, University of Southern California, Aug. 1978.

This report discusses current and future problems and benefits of the mathematical proof of program properties, i.e., verifying program properties against a specification of expected behavior. It is argued that the related theory contributes to our knowledge of program construction and maintenence. The author argues that neither proving nor testing programs is adequate on its own as a verification technique but rather that one method or the other applied in recognizably acceptable situations, both methods in parallel, or some combination of the two may produce the best results.

Some current needs are: expanding the theory to encompass more aspects of program correctness, increased effectiveness of the theory for program verification, and the development of the human and technical aspects of program specification and verification.

Breakthroughs needed for making program proving a normal activity are: a coherent connection with program testing, ability to reuse and adapt existing proofs, a methodology for evaluation proof techniques, improved mechanical theorem proving, large-scale demonstrations, and improved expressibility and readability.

Benefits claimed for program verification are: higher quality programs, a deeper understanding of programs, formal reasoning methods for construction, and a complement to program testing.

A large example verification is given in an appendix to illustrate the generality and variety of techniques.

Gerritsen78
Gerritsen, R. Steps towards the automation of database design. In NYU78.

Geschke75
Geschke, C. and Mitchell, J. On the problem of uniform references to data structures. *IEEE Trans. Soft. Eng. SE-1*, 2 (June 1975), pp. 207-219.

The cost of a change to a large software system is often primarily a function of the size of the system rather than the complexity of the change. One reason for this is that programs which access some given data structure must operate on it using notations which are determined by its exact representation. Thus, changing how it is implemented may necessitate changes to the programs which access it. This paper develops a programming language notation and semantic interpretations which allow a program to operate on a data object in a manner which is dependent only on its logical or abstract properties and independent of its underlying concrete representations. (authors' abstract)

Geschke78
Geschke, C. M., Morris, J. H. and Satterthwaite, E. H. Early experiences with MESA. *Comm. ACM 20*, 8 (Aug. 1978), pp. 540-553.

The experiences of Mesa's first users - primarily its implementors - are discussed, and some implications for Mesa and similar programming languages are suggested. The specific topics addressed are: module structure and its use in defining abstractions, data-structuring facilities in Mesa, an equivalence algorithm for types and type coercions, the benefits of the type system and why it is breached occasionally, and the difficulty of making the treatment of variant records safe. (authors' abstract)

Goas78
Goas, G. and Kastens, U. Programming languages and the design of modular programs. In Hibbard, P. and Schuman, A (Eds.), *Constructing Quality Software*, North-Holland, New York, 1978.

Goguen77
Goguen, J. A., Thatcher, J. W., and Wagner, E. G. An initial algebra approach to the specification, correctness, and implementation of abstract data types. RC 6487, IBM Yorktown Heights, New York, April 1977, and in Yeh77a.

Abstract data types have been claimed a powerful tool in programming (as in SIMULA and CLU), both from the viewpoint of user convenience, and that of software reliability. Recently algebra has emerged as a promising method for the specification of abstract data types; this makes it possible to prove the correctness of implementations of abstract types. It also raises the question of the correctness of the specifications, and the proper method for handing run-time errors in abstract types. Unfortunately not all of the algebra underlying these issues is entirely trivial nor has it been adequately developed or explained. This paper shows how a reasonable notation for many-sorted algebras makes them just about as manageable as one-sorted (universal) algebras, and presents comparatively simple yet completely rigorous statements of the major algebraic issues underlying abstract data types. We present a number of specifications, with correctness proofs for some; the issue of error messages is thoroughly explained, and the issue of implementations is broached. (authors' abstract)

Goguen78
Goguen, J. A. Abstract errors for abstract types. In Neuhold, E. (Ed.), *Formal Description of Programming Concepts*. North-Holland, 1978, pp. 491-526.

In order that an abstract data type specification shall in fact encapsulate the intended semantics, it is essential that it also specify how exceptional states shall be handled, including any error messages that shall be produced in response to exceptional conditions. Thus, abstract types must include abstract errors. This paper treats abstract errors within the algebraic framework for data abstraction, introducing the notion of an "error algebra." It is shown how to treat the basic problems of specification, implementation, correctness, and parameterization, for abstract types with abstract errors. A number of examples illustrate the basic points, and a rigorous mathematical theory is provided. A programming and specification language called OBJ embodying these features is introduced and used for examples. The appendix discusses the implementation of OBJ. (author's abstract)

Goguen79a
Goguen, J. A., Tardo, J. J., Williamson, N. and Zamfir, M. A practical method for testing algebraic specifications. *UCLA Computer Science Dept. Quarterly*, 1979, pp. 59-80.

Goguen79b
Goguen, J. A. and Tardo, J. J. An introduction to OBJ: A language for writing and testing formal algebraic program specifications. In PSRS79.

OBJ is a formal language for writing and testing algebraic program specifications; it is also an applicative, non-procedural programming language. Specifications are tested for consistency with their intended initial algebra semantics by means of a rewrite-rule based operational semantics. OBJ has several unconventional features designed to make algebraic specifications more expressive and better organized, and hence easier to read, write, and verify. These include a flexible syntax for user-defined operations, error (exception) conditions, methods for decomposing specifications into smaller "mind size" pieces, and an optimal evaluation strategy which terminates for a rather large class of specifications, including those (such as a SET) with commutative equations. A number of examples and some particulars of the current implementation are presented. Some features planned for more advanced implementations are also described. (authors' abstract)

Goldberg76
Goldberg, and Kay, A. SMALLTALK-72 Instruction Manual, SSL 76-6, Xerox PARC, 1976.

Goldman79
Goldman, N. and Wile, D.S. A relational data base foundation for process specifications. In Chen79.

A language suitable for system specification should allow a specification to be based on a cognitive model of the process being described. In part, such a language can be obtained by properly combining certain conceptual abstractions of entity-relationship data models with reference and control concepts designed for programming languages. Augmenting the resulting language with formal versions of several natural language constructs, such as temporal reference, further decreases the cognitive distance between specifications of large systems and the modelled world. (author's abstract)

Goldsmith74
Goldsmith, C. The design of a procedureless programming language. *SIGPLAN Notices 9*, 4 (1974), pp. 13-24.

Good78
Good, D. I., Cohen, R. M., and Hunter, L. W. A report on the development of Gypsy. *Proc. 1978 ACM Annual Conf.*, Washington, D. C., Dec. 1978, pp. 116-122.

The authors describe the changes introduced in the second version of Gypsy, an extension of PASCAL, which supports the specification and construction of verified programs. The language was developed together with an integrated methodology for specifying, programming, and verifying systems. Major features of Gypsy now include a specification language and a programming language, which supports concurrency, user programmable exception handling, lemmas, dynamic storage allocation, and incremental verification.

There is a strong emphasis on the specification, programming, and verification of individual Gypsy programs. These program descriptions are stored in a "database" that supports incremental verification, i.e., changes that affect existing programs are noted to be reproven.

The Gypsy compiler version 2.0 is not yet completed

Goodenough75
Goodenough, J. B. Exception handling: Issues and proposed notation. *Comm. CACM 18*, 12 (Dec. 1975).

GREEN78
Rationale for the design of the GREEN Programming Language, submitted to the U.S. Department of Defense, Feb. 1978.

Gries77
Gries, D. and Gehani, N. Some ideas on data types in high-level languages. *Comm. ACM 20*, 6 (1977).

A number of issues are explored concerning the notion that a data type is a set of values together with a set of primitive operations on those values. Among these are the need for a notation for iterating over the elements of any finite set (instead of the more narrow *for* i := 1 *to* n notation), the use of the domain of an array as a data type, the need for a simple notation for allowing types of parameters to be themselves parameters (but in a restrictive fashion), and resulting problems with conversion of values from one type to another. (authors' abstract)

Guide70
Guide-Share Data Base Management System Requirements. Joint Guide-Share Database Requirements Group, Guide-Share, New York, 1970.

Guttag75
Guttag, J. V. The specification and application to programming of abstract data types. Ph.D. diss. CSRG-59, University of Toronto, Sept. 1975.

Abstract data types can play a significant role in the development of software that is reliable, efficient, and flexible. Unfortunately, the techniques currently available for specifying abstract data types are inadequate. The techniques available within programming languages fail to provide a mechanism for disentangling the abstract meaning of a data structure from a particular representation of it. Many extra-programming-language techniques, e.g., operational semantics, also exhibit this failing. A second common failing of extra-programming-language approaches is a lack of formalism. This tends to lead to specifications that are inconsistent or ambiguous.

This thesis presents an algebraic specification technique that overcomes these problems. It is derived from Hoare's work on the axiomatic specification of the semantics of programming languages, and from Birkhoff and Lipson's heterogeneous algebras. An algebraic specification of an abstract data type consists of two parts: a syntactic specification and a set of relations. The syntactic specification provides the names, ranges, and domains of the operations associated wth the type. The relations define the meaning of the operations by stating their relationships to one another.

The prime difficulty in constructing algebraic specifications of abstract data types lies in ascertaining whether or not a specification is sufficient to fully define the type. For this reason, a mechanical technique for verifying the sufficient-completeness of certain classes of algebraic specifications has been devised.

Other discussions include a treatment of some of the theoretical properties of the specification technique presented, and discourses on the application of the algebraic specification of abstract data types to top-down program design and program verification. (author's abstract)

Guttag76
Guttag, J. V., Horowitz, E. and Musser, D.R. The design of data type specifications. In PSE76 and in Yeh77a.

This paper concerns the design of data types in the creation of a software system; its major purpose is to explore a means for specifying a data type that is independent of its eventual implementation. The particular style of specification, called algebraic axioms, is exhibited by axiomatizing commonly used data types. These examples reveal a great deal about the intracacies of data type specification via algebraic axioms, and also provide a standard to which alternative forms may be compared. Further uses of this specification technique are in proving the correctness of implementations and in interpretively executing a large system design before actual implementation commences. (authors' abstract)

Guttag77a
Guttag, J. V., Horowitz, E. and Musser, D.R. Some extensions to algebraic specifications. *SIGPLAN Notices 1*, 3 (March 1977).

Algebraic specifications of abstract data types are beginning to gain wide currency. In this paper the authors discuss an extension to this specification technique which allows the specification of procedures which alter their parameters, and various ways of handling the specification of error conditions. (authors' abstract)

Tentative solutions are offered for the two problems addressed. The specification of error conditions or partial functions is done by restriction specifications that define when the value of an operation is not well defined. The problem of specifying procedures which alter their parameters was seen as more difficult. The authors conclude that pure functions are impractical but that Euclid's solution of no aliasing, i.e., procedures can refer objects only through formal parameters, is not too restrictive and sufficient.

Guttag77b
Guttag, J. V. Abstract data types and the development of data structures. *Comm. ACM 20*, 6(June 1977), pp. 396-404.

Abstract data types can play a significant role in the development of software that is reliable, efficient, and flexible. This paper presents and discusses the application of an algebraic technique for the specification of abstract data types. Among the examples presented is a top-down development of a symbol table for a block structured language; a discussion of the proof of its correctness is given. The paper also contains a brief discussion of the problems involved in constructing algebraic specifications that are both consistent and complete. (author's abstract)

Guttag78
Guttag, J. V. and Horning, J. J. The algebraic specification of abstract data types. *Acta Informatica 10*, 27-52 (1978).

Guttag79a
Guttag, J. V. Notes on type abstraction. In PSRS79.

This tutorial paper begins by discussing the role of type abstraction and the need for their formal specification. It examines two major approaches to type specification: Hoare's axiomatic approach (as embodied in EUCLID) and Guttag's algebraic approach. In addition to covering older material, new aspects such as parameterized types and the specification of restrictions are discussed. It is argued that facilities for abstract types be included in programming languages to aid program design and understanding, and to permit (strong) type checking. Both techniques provide a basis for the verification of programs which permits the factoring of the proof. The author concludes that neither technique is uniformly best and that discussions of the relative merits must be highly subjective. (partly from author's abstract)

Guttag79b
Guttag, J. V., Horowitz, E. and Musser D.R. Abstract data types and software validation. *Comm. ACM 21*, 12 (Dec. 1978), pp. 1041-1063.

A data abstraction can be naturally expressed using algebraic axioms, whose virtue is that they permit a representation-independent formal specification of a data type. A moderately complex example is given which shows how to employ algebraic axioms at successive levels of implementation. The major thrust of the paper is twofold. First, it is shown how the use of algebraic axiomatizations can significantly improve the process of proving the correctness of an implementation of an abstract data type. Second, semi-automatic tools are described which can be used both to automate such proofs of correctness and to derive an immediate implementation from the axioms. This implementation allows for limited testing of a large software system at design time, long before a conventional implementation is accomplished. Extensions to the formalism which permit the handling of errors and procedures with side effects are also given. (authors' abstract)

Hammer75
Hammer, M. and McLeod, D. J. Semantic integrity in a relational database system. In VLDB75.

The authors propose a semantic integrity subsystem for describing and preserving database semantic integrity. Semantic integrity is discussed as an aspect, along with reliability, concurrent consistency, and security, of the problem of preserving database accuracy, correctness, and validity. The authors adopt the state snapshot approach in such a way as to capture state transition constraints.

The semantic integrity subsystem is to have five components: 1) a high level language to define the nature of the constraint, the enforcement conditions and the violation actions; 2) a constraint language processor; 3) a constraint enforcer; 4) a violation-action processor; and 5) a constraint compatibility checker. The authors describe and give examples of aspects of the relation constraint language.

Hammer76a
Hammer, M. M. Data abstraction for databases. In Utah76.

The author describes data abstraction as emerging from programming language research as a tool for modular, structured programming. Data abstraction is used to separate the description of a data object from its implementation, hence users may concentrate on the behavioral semantics (i.e., the meaningful operations) of the type.

Although the principle of abstraction and that of data independence are similar, the former focuses on clarity and modifiability while the latter focuses on the logical-physical separation. For example, the ANSI/SPARC framework can be viewed as layers of abstraction and the relational database model supports high level views of data.

A major difference between data abstraction and high level data models is that data structures are representational and the semantics of abstract data types are entirely behavioral. Advantages of using abstract data types in databases include: constrained access via meaningful operations, potential to alter representations, and controlling semantic integrity.

Problems in using data abstractions in databases include: top-down development versus data-up design of database applications; the need for meaningless operations, e.g., for error correction; a fixed set of operations is too restrictive; isolation and cooperation of abstractions; specialization of behavior specifications in different views.

This early paper did not recognize the fact that only modification operations need be fixed and that modification operations are as static as the structures (at last at the conceptual level) they operate on.

Hammer76b
Hammer, M. M. and McLeod, D. A framework for database semantic integrity. In PSE76.

The authors present a structured framework for describing the semantic integrity requirements of relational databases. Four levels of semantic integrity information are described: domain definitions, relation structure, structured operations, and relation constraints. They address relation constraints which have three component parts: an assertion, a validity condition (when it applies), and a violation-action. The authors deal primarily with the properties of relation constraint assertions. Simple assertions delimit some data (constrained) by other data (constraining). Derived assertions are Booleans of simple assertions. Assertions are classified and the classification is used to develop a constraint language. Examples of constraints are expressed in the language.

Hammer77
Hammer, M., Howe, W.G., Kruskal, V.J. and Wladawsky, I. A very high level programming language for data processing applications. *Comm. ACM 20*, 11(Nov. 1977).

Application development today is too labor-intensive. In recent years, very high-level languages have been increasingly explored as a solution to this problem. The Business Definition Language (BDL) is such a language, one aimed at busness data processing problems. The concepts in BDL mimic those which have evolved through the years in businesses using manual methods. This results in three different sublanguages or components: one for defining the business forms, one for describing the business organization, and one for writing calculations. (authors' abstract)

Hammer78a
Hammer, M. M. and McLeod, D. The semantic data model: a modelling mechanism for data base applications. In SIGMOD78.

Conventional data models are not satisfactory for modelling database application systems. The features that they provide are too low level and representational to allow the semantics of a database to be directly expressed in the schema. The semantic data model (SDM) has been designed as a natural application modelling mechanism that can capture and express the structure of an application environment. The features of the SDM correspond to the principal intensional structures naturally occurring in contemporary database applications. Furthermore, facilities for expressing derived (redundant) information are an essential part of the SDM; derived information is as prominent in an SDM schema as is primitive data.

The SDM is designed to enhance the effectiveness and usability of computerized database design process, can serve as a formal specification and documentation mechanism for a database, and can also support a variety of powerful user interface facilities. (authors' abstract)

Hammer78b
Hammer, M. and Sarin, S. K. Efficient monitoring of database assertions. In SIGMOD78.

Hammer79
Hammer, M. and McLeod, D. On database management system architecture. Technical Report, LCS, MIT, 1979.

The authors examine some basic assumptions of the two decade old DBMS paradigm and propose some alternatives. They view a DBMS as the central part of an information system rather than a support facility which provides "fancy" access methods. Two main proposals are made. First, the integration of programming language and database concepts so as to provide full language capabilities (control and data structures, exception handling, full computational power) over databases. Second, a federation of loosely coupled databases as opposed to a centralized schema with little local autonomy for individual applications.

The first proposal is expressed in terms of an extended schema concept. Principally, their schema is to include knowledge currently embedded in procedures so that both the static and the dynamic semantics of applications are defined. Specifically, schemas are to include: logically-oriented, representation-free descriptions; a minimal gap between conceptual and external schemas; relativist views that permit alternate interpretations; entity naming using the entity itself rather than values of specific attributes; extensible conceptual schema; subtypes; multiply typed entities; less distinction between data and meta-data; and support for derived information.

Hantler76
Hantler, S. L. and King, J. C. An introduction to proving the correctness of programs. *ACM Computing Surveys 8*, 3 (Sept. 1976).

This paper explains, in an introductory fashion, the method of specifying the correct behavior of a program by the use of input/output assertions and describes one method for showing that the program is correct with respect to those assertions. An initial assertion characterizes conditions expected to be true upon entry to the program and a final assertion characterizes conditions expected to be true upon exit from the program. When a program contains no branches, a technique known as symbolic execution can be used to show that the truth of the initial assertion upon entry guarantees the truth of the final assertion upon exit. More generally, for a program with branches one can define a symbolic execution tree. If there is an upper bound on the number of times each loop in such a program may be executed, a proof of correctness can be given by a simple traversal of the (finite) symbolic execution tree.

However, for most programs, no fixed bound on the number of times each loop is executed exists and the corresponding symbolic execution trees are infinite. In order to prove the correctness of such programs, a more general assertion structure must be provided. The symbolic execution tree of such programs must be traversed inductively rather than explicitly. This leads naturally to the use of additional assertions which are called "inductive assertions." (authors' abstract)

Hayes74
Hayes, P. J. Some problems and non-problems in representation theory. *Proc. AISB Summer Conf.*, Univ. of Sussex, July, 1974.

Hayes77
Hayes, P. J. In defense of logic. *Proc. 5th Int'l Joint Conf. on Artificial Intelligence*, MIT, 1977.

Hendrix75
Hendrix, G. G. Expanding the utility of semantic networks through partitioning. *Proc. of Int'l. Joint Conf. on Artificial Intelligence*, Sept. 1975, Tbilisi, USSR.

Hewitt72
Hewitt, C. Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. Ph.D. thesis, Dept. of Math., MIT, 1972.

Hewitt73
Hewitt, C. et al. A universal modular ACTOR formalism for artificial intelligence. *Proc. of the 3rd IJCAI*, 1973.

Hoare72a
Hoare, C. A. R. Notes on data structuring. *APIC Studies in Data Processing* No. 8: *Structured Programming*, Academic Press, New York, 1972.

The author motivates the monograph by describing abstraction as "arising from the recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate on these similarities, and to ignore for the time being the differences." Illustrations are drawn from the real world and from programming. He then proceeds to develop the concept of type from unstructured types to types structured under the disciplines of cartesian product, disciminated union, the array, powerset, sequence, and recursion. Each structure is described in terms of its meaning, manipulations, and representations. Finally, the notion of axiomatization of types is presented.

Hoare72b
Hoare, C. A. R. Proof of correctness of data representation. *Acta Informatica 1*, pp. 271-181 (1972).

Hoare proposes that one should develop algorithms with abstract data concepts, and choose a representation for the data after the design is finished. In this paper, he discusses the problem of proving that the concrete representation chosen actually performs to the requirements of the abstract concept. He uses the SIMULA data constructs and provides a small example proof. Functions with side effects are also discussed.

Hoare72c
Hoare, C. A. R. The quality of software. *Software - Practice and Experience 2*, pp. 103-105 (1972).

The main problem in the design of any engineering product is the reconciliation of a large number of strongly competing objectives. In the case of general-purpose computer software, I have made a list of no less than seventeen: 1) clear definition of purpose; 2) simplicity of use; 3) ruggedness; 4) early availability; 5) reliability; 6) extensibility and improvability in light of experience; 7) adaptability and easy extension to different configurations; 8) suitability to each individual configuration of the range; 9) brevity; 10) efficiency (speed); 11) operating ease; 12) adaptability to a wide range of applications; 13) coherence and consistency with other programs; 14) minimum cost to develop; 15) conformity to national and international standards; 16) early and valid sales documentation; and 17) clear, accurate and precise user's documents.

There can be little doubt that modern software fails to reconcile these objectives, particularly with respect to one or more of (1) - (6), (8) - (10), (14), and (17). What is the solution? The first necessity is for the software designer and his customers to recognize that there is a conflict of objectives,

and that there is the need for compromise. Secondly, it is essential that the compromise be selected in the light of a sound knowledge of good algorithms and program structures, and not merely as a result of wishful thinking of a sales or product specification department.

. . .Unfortunately, in the past decade, software writers' main pride has been to implement an arbitrary specification handed down to them from "above" (e.g., ALGOL68, PL/1); and not to pay much concern to the quality of either the specification or the product. They tend to argue from the fact that you can do anything by software to the conclusion that you can do everything by software. And you can, too; but is it worth it? Only very few software designs (a notable example being PASCAL) have actually been made to optimize the quality of the product in the light of known software techniques.

So my advice to the designers and implementors of software of the future is in a nutshell: do not decide exactly what you are going to do until you know how to do it; and do not decide how to do it until you have evaluated your plan against all the desired criteria of quality. And if you cannot do that, simplify your design until you can. (CR23, 801, From the Guest Editorial)

Hoare73a
Hoare, C. A. R., and Wirth, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica 2*, pp. 335-355 (1973).

An attempt to describe formally the semantics of the language PASCAL. The basis for this is the approach expounded in Hoare's 1969 article "An Axiomat Axiomatic Basis for Computer Programming". The definition is quite useful but it is incomplete: some language features were left out and some restrictions were made. These omissions seem to be due to complexity and point to parts of the language worth considering. The omissions include: real arithmetic, GO TOs, ALPHA-type. The following features are imperfectly described: classes, procedures, and functions with regard to global variables. This document does not replace the original report, it only supplements it. (D. Barnard)

Hoare73b
Hoare, C. A. R. Recursive data structures. Stanford University, Computer Science Dept., STAN-CS-73-400 (Oct. 1973), pp. 1-32.

The power and convenience of a programming language may be enhanced for certain applications by permitting tree-like data structures to be defined by recursion. This paper suggests a pleasing notation by which such structures can be declared processed; it gives the axioms which specify their properties, and suggests an efficient implementation method. It shows how a recursive data structure may be used to represent another data type, for example, a set. It then discusses two ways in which significant gains in efficiency can be made by selective updating of structures, and gives the relevant proof rules and hints for implementation. The examples show that a certain range of applications in symbol manipulation can be efficiently programmed without introducing the low-level concept of a reference into a high-level programming language. (from author's abstract)

It appears to this reviewer that over the next few years there will begin to appear new programming languages in which the description of algorithms will take second place to the description of data structures, and that this is indeed a desirable direction in which to migrate. We seem to have some properties of control structures well in hand, due to the developments of Dijkstra, Mills, et al.; in this paper the author points out similarities in the realm of data structures, and shows that the same general tools used in describing (recursive) control structures are applicable to the description of data structures in recursive fashion.

The paper is well worth reading, and strongly recommended to language/compiler designers and implementors. (CR29, 165 L. D. Yarbrough)

Hoare74
Hoare, C. A. R., and Lauer, P. E. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica 3*, 135-153(1974).

This paper presents a comparative study of different methods for formal description of programming languages. These methods have been applied to a simple but realistically usable programming language; the more abstract definitions have been proved to be consistent relative to the more concrete ones.

The authors argue that a single language be defined formally using several different methods, each definition oriented towards the needs of a particular class of user. Once defined, it is important to prove that they define the same language.

Hoare75
Hoare, C. A. R. Data reliability. In PSE75 and in Yeh77a.

The author surveys problems in achieving data reliability and finds them more severe than those, already notorious, of program reliability. Reasons for this are: 1) programs are analyzable but data is extremely difficult to interpret; 2) program errors can be isolated but data errors tend to propagate; 3) failed programs can be re-run but may leave errors in a database; 4) errorneous databases destroy user confidence.

The author outlines conceptual and methodological tools which are available to solve these problems. The primary concept is that of type. By analogy with control structure disciplines, desirable features of structuring disciplines are given: few in number, logically simple, amenable to proof, applicable on large and small scales, support top-down or bottom-up design, easy and efficient to implement. It is argued that the disciplines of direct product, union, sequence, recursion, and mapping are "complete" (can describe any data structure) and fulfill the criteria.

It is argued that databases cannot be designed by purely top-down or bottom-up but that abstraction is a major design tool. He argues for the elimination of pointers which are analagous to "goto's".

Holt78
Holt, R. C., Wortman, D. W., Cordy, J. R., and Crowe, D. R. The Euclid Language: A progress report. *Proc. 1978 ACM Annual Conf.*, Washington, D. C., Dec. 1978, pp. 111-115.

This paper gives a summary of language changes, the status of the implementation, and some observations about the use of Euclid as a practical programming language. Euclid is a language for writing verifiable system programs. Euclid is a direct extension of PASCAL and includes: explicit importing and exporting of identifiers from modules; extending types to have parameters so that a type declaration can be a template for many different instance types; module types to provide data abstraction and information hiding; and assert statements.

The authors describe language changes needed to resolve problems encountered during the compiler implementation. They concern: legality assertions, type compatibility and conversion, variant records and discriminating cases, strings, well-behaved arithmetic, uninitialized variables, and type parameterization.

The authors' experience with a subset of Euclid lead them to believe that it can be a practical production software tool, even when formal verification is not attempted.

Holt79
Holt, R.C. and Wortman, D.B. A model for implementing Euclid modules and type templates. *SIGPLAN Notices 14*, 8(1979).

Horning76
Horning, J. J. Some desirable properties of data abstraction facilities. In Utah76.

The author describes data abstraction facilities in terms of capsules (a data analog to procedures): their advantages, what they are, and what they are not.

Eight advantages are that capsules support: (1) repetition - define once, use often; (2) modularity - decomposition of complex structures and information hiding; (3) structured programming - modularize and record design decisions; (4) conceptual units - smaller units which emphasize *what* rather than *how* or *why* and increase the feasibility of proofs and understanding; (5) specification - ability to specify abstract properties of structure precisely; (6) maintenance - isolate changes and errors; (7) language extension - use capsules to add new types "to the language"; and (8) separate compilation of capsules.

A capsule consists of a name, an interface specification, and a body. The name together with parameters permits the creation of instances. The interface provides the names and parameters of the only operations available on capsule instances. The body contains the implementation. Capsules are not procedures. Procedures are invoked and terminate before the next procedure executes. An instantiation of a capsule is an object that persists in its scope even with no active operations. Capsules cannot define new data structures just as procedures cannot define new control structures.

Research issues in this area include: parameter passing, return results, environment inheritance, checkable redundancy (specifications), proof techniques, and notations.

IBM
Data Base Design Aid: Designer's Guide. IBM #GH20-16.

Ingalls78
Ingalls, D. The Smalltalk-76 programming system: Design and implementation. *Proc. 5th POPL*, Tucson, Arizona, 1978, pp. 9-16.

Irwin75
Irwin, J. and Srinivasan, C. Description of CASNET in MDS, CBM-TR-49, Rutgers University, 1975.

Jackson75
Jackson, M. A. *Principles of Program Design*. Academic Press, New York, 1975.

Johnson76
Johnson, R. T. and Morris, J. B. Abstract data types in the MODEL programming language. In Utah76.

Jones78
Jones, A. K. and Liskov, B. H. A language extension for expressing constraints. *Comm. ACM 21*, 5 (May 1978).

Jones79
Jones C. A survey of programming design and specification techniques. In PSRS79.

The author surveyed programming design methods and found many different approaches to viewing business or technical problems and 150 design languages. The approaches emphasize functional analysis or data analysis. Functional analysis is the oldest technique and involves examining functions to be embodied in the result (e.g. top down design, bottom up design, composite/structured design). Data analysis is newer and concerns the analysis of data the system will use (e.g. Jackson's and Warnier's methodologies). The author claims that these two diverse approaches cannot be integrated. The author also discusses design languages and the problems they are used to solve.

Katz79
Katz, R. and Wong, E. Heterogeneous data models - Part 1: Semantic Issues. ERL memo UCB/ERL m79/56, U.C. Berkeley, August 1979.

Kent78
Kent, W. *Data and Reality*. North-Holland Publ. Co., New York, 1978.

Kent79
Kent, W. Limitations of record-based information models. *ACM TODS 4*, 1 (March 1979).

King79
King, J.C. Program correctness: On inductive assertion methods IBM RJ2525, May 1979.

A study of several of the proof of correctness methods is presented. In particular, the form of the induction used is explored in detail. A relational semantic model for programming languages is introduced and its relation to predicate transformers is explored. A rather elementary viewpoint is taken in order to expose, as simply as possible, the basic differences of the methods and the underlying principles involved. These results were obtained by attempting to thoroughly understand the "subgoal induction" method. (author's abstract)

Kowalski74
Kowalski, R. Logic as a programming language. *Proc. Info. Processing 74*, North-Holland, 1974.

Kowalski76
Kowalski, R. Algorithms = Logic + Control. Internal publication, Imperial College, London, 1976.

Lampson77
Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. J. Report on the programming language EUCLID. *SIGPLAN Notices 12*, 2 (Feb. 1977).

This report describes the programming language Euclid which is intended for the expression of system programs which are to be verified. Euclid is based strongly on PASCAL but also borrows from Alphard, BCPL, CLU, Gypsy, LIS, Mesa, Modula, and SUE. A verifiable program is one written in such a way that existing formal techniques for proving certain properties of programs can be readily applied; proofs might be automatic or manual.

Euclid differs from PASCAL mostly by restriction as follows: visibility of identifiers by explicit importation and exportation of identifiers; a notion of "same type" and scope; restricting pointers; explicit control of storage allocation for dynamic variables; parameterized types; records with constant components; modules which can contain routine and type components and mechanisms for initialization and finalization; for statement to enumerate a sequence of values; a mechanism with which to use features of the underlying machine; and assertions.

A novel feature of Euclid is the intent to generate automatically legality assertions which are statements that could not be verified automatically but which must be verified in order to render the program "legal."

Lampson79
Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G., and Popek, G. J. *Revised report on the programming language EUCLID*, Xerox Palo Alto Research Center Tech. Report CSL78-2.

Langefors77
Langefors, B. Information systems theory. *Information Systems 2*, pp. 209-219 (1977).

Ledgard77
Ledgard, H. F. and Taylor R. W. Two views of abstraction. *Comm. ACM 20*, 6 (June 1977).

This paper briefly reviews the 1976 Utah. Conference on Data: Abstraction, Definition, and Structure which had the purpose of promoting and bringing together the relatively disjoint areas of database and programming language research over their mutual concern for data abstraction. The authors view data abstraction as a means for concentrating on the essential aspects of data rather than on physical details. A primary goal of data abstraction is to provide intellectually manageable access to complex systems.

The emphasis in programming language research has been on problem oriented programming, representation independence, type definition mechanisms, verification of data abstractions, and the need for new control structures. Little research has addressed the issues of interrelationships between data abstractions, and mechanisms for dealing with complex interrelated files.

Database research has been concerned with database design, application evolution, non-procedural languages, and data reliability. Sheer complexity is the main problem. However, there has been little discussion of programming language extensions and disciplines to aid in the development, verification, and maintenance of programs.

Neither area can exist without the other. The main questions that remain are: what is the nature of data abstraction? How are systems decomposed into abstractions? Data abstraction appears to show great promise but progress has been slow. The authors did not remark on the lack of communication between people in the two areas. The conference has been characterized as two conferences meeting in the same room.

Leavenworth74
Leavenworth, B. and Sammet, J. An overview of nonprocedural languages. *SIGPLAN Notices 9*, 4 (1974), pp. 1-12.

Leavenworth79a
Leavenworth, B. M. On the construction of non-procedural programs. RC 6155, IBM Yorktown Heights, May 1976.

Non-procedural programming involves raising the level of algorithm description by adding domain specific operators to a base language. This paper proposes an applicative language as a suitable basis for describing business applications, after the appropriate operators have been defined. The key features of the language and the domain specific operators are introduced. An example is used to discuss a methodology for constructing non-procedural programs for business applications.

Leavenworth79b
Leavenworth, B. The use of data abstraction in program design. RC 7637, IBM Research, June 1979.

This paper proposes a design language based on PL/1, called extended PL/1, which supports data abstraction. The language is strongly typed so that consistency can be checked automatically. It has parameterized data types for data abstraction and generic procedures for procedural abstraction. A generic procedure defines a class of procedures that can work on a variety of data types. The language has an external structure mechanism to be used to specify module interfaces. Two types of modules are supported: procedures that accept abstract data objects as parameters and (optionally) return abstract data objects. Capsules allow the definition of abstract data types and support information hiding.

The author claims that the language supports: abstraction, information hiding, modularity, module interface specification, integration with the Jackson design methodology, and the construction of a database of abstractions.

LeVanKiet78
Le Van Kiet. *The Module: A tool for structured programming*, Ph.D. Diss. ETH Nr. 6153, Zurich, 1978.

This thesis presents a complete and systematic study of the module concept. Modular programming is intuitively known and currently used by most programmers. However, the modularization process will never become a well-defined and practical discipline if programming languages do not support a corresponding construct. Techniques of representing the module concept in programming languages are discussed; a complete demonstration of the Modula module is done in writing a compiler in Modula, the language it compiles. Progress in the development of the theory of structured programming could be achieved only by means of a fundamental evolution of structures supporting these concepts within programming languages. (from author's abstract)

The thesis contains an excellent survey of the module concept. The author gives a testimonial to the module concept after extensive experience in programming with modules. He demonstrates their usefulness for stepwise development of systems, understandability, correctness, reliability, and portability. He proposes that 90 percent of all modules found in practice export no type and are used for information hiding. Only 10 percent export types and operators for the purpose of data abstraction. The author contends that the module concept can be useful and practical for all programmers.

Levesque77
Levesque, H. A procedural approach to semantic networks. TR-105, University of Toronto, 1977.

This thesis investigates some of the issues involved in modelling a domain of knowledge. A representation based on semantic networks is presented, where the semantics are specified by procedural attachment. In addition, programs are integrated directly into the representation allowing it to be completely self-contained. A methodology is then proposed for organizing and structuring the model hierarchically.

Related issues, such as inheritance and visibility, are also discussed. Finally, examples of the use of the representation are presented, includng a formal specification of the representation in terms of itself. (author's abstract)

Levesque79
Levesque, H. and Mylopoulos, J. A procedural semantics for semantic networks. In Findler79.

Linden76
Linden, T. A. The use of abstract data types to simplify program modifications. In Utah76.

The paper provides an introduction to abstract data type (ADT) concepts and motivates their use as units of modularity. The thesis of the paper is that in using ADT's as the basic unit of modularity in a program, the program will be easier to modify and extend due to module independence. ADT's are viewed as a vehicle for data abstraction which is a generalization of both data types and procedures.

The author claims that a data management system is an ADT and that ADT's can be used to improve reliability, security, and data structure modification in data management systems.

A survey of existing ADT facilities and uses is given. Some advantages given are: restricting access to data structures via fixed operations; a unit of modularity for systems using Dijkstra's hierarchical structure; documentation of design decisions; and compatibility with existing specification techniques. The author encourages the use of ADT disciplines even if the target language does not support them. He notes that ADT concepts, uses, and facilities are still very volatile.

The thesis of the paper is illustrated by an example in which a program evolves without losing the structure developed in the first stage.

Linden78
Linden, T. A. Specifying abstract data types by restriction. *SIGSOFT SEN 3*, 2 (April 1978).

The author introduces the notion of type restriction to be used to simplify program specifications and to take advantage of similarities that exist between different abstract data types. The concept of a type restriction is borrowed from mathematical logic. A type restriction has the semantics of the original type with the exception of one or more functions that are removed from the original specification and become hidden functions. In this way the resultant type is restricted to a subset of the properties of the original type. It is argued that restriction provides a theoretical justification of the concept of hidden functions that appear in some current specification techniques. A traversible stack is used to demonstrate the simplification of specifications by restriction.

Lind79
Lindencrona-Ohlin, E. A study of conceptual data modeling, Ph.D. Diss. Dept. Computer Sciences, Chalmer's University of Technology, Goeteborg, Sweden.

The conceptual level of data base systems was introduced by the ANSI/SPARC Interim Report 1975. During the late 1970's a number of data models and design methods intended for the conceptual level have been suggested.

There does not exist any common agreement on the scope and contents of the conceptual level design area. In this study, an attempt is made to define informally the contents of conceptual level design in terms of problem areas.

Further, an attempt is made to identify and discuss semantical aspects of conceptual level data base design. For this purpose a number of design methods are analyzed. This analysis aims at identification of relevant aspects and problems within the area. Important aspects identified and discussed concern; inference, redundancy and a temporal dimension. Among these aspects, inference is most thoroughly discussed. Different types of inference are identified. Redundancy is considered closely related to inference. Requirements of "non-redundancy" in conceptual level models is questioned.

This study contributes to the development of conceptual level models and methods, and to further approaches to classification of such methods. (author's abstract)

Liskov72
Liskov, B. H. A design methodology for reliable software systems, *Proc. of the FJCC 41* (1972), pp. 191-199.

This paper presents a methodology for the development of reliable software. It begins by justifying the development of such a methodology in light of the failure of existing methods (involving extensive debugging) to produce reliable software. The author then describes a two-part methodology derived from her own experience with a large software project.

The first part involves the definition of a "good" system modularization, in which the system is organized into a hierarchy of "partitions," each corresponding to a level of abstraction, and having minimal connections with one another. The total system design is then expressed as a structured program, rendering the design amenable to existing proof techniques.

The second part looks at the question of how to achieve a system design with good modularity. The key to the design is seen by the author as the identification of "useful" abstractions which help the designer to think about the system. Some techniques for finding such abstractions are given. A definition of "end of design" is given, involving having a system design with the desired structure, and a preliminary user's guide.

The paper ends by describing experiments in the use of the methodology in progress at the time of presentation. (CR25, 795)

Liskov74
Liskov, B. H. and Zilles, S. N. Programming with abstract data types. *SIGPLAN Notices 9*, 4 (April 1974).

The authors describe the concept of abstraction, i.e., concentrating on relevant details and suppressing irrelevant details, as a basis of high level problem-oriented programming. It is argued that functional abstraction supports structured programming techniques such as successive decomposition allowing programs to build abstractions from existing abstractions. The resulting modules: procedures (functional abstractions) and operation clusters (data types and their legal operations), enhance reliability, understandability, maintainability, and correctness proofs. The main contribution of this paper is the operation cluster concept which provides a user interface to objects of a given data type and restricts access to these objects through given names and operations. Clusters provide information hiding, strong type checking, and potential for compile time and execution efficiencies. These features are illustrated using programming examples.

Liskov75
Liskov, B. H. and Zilles, S. N. Specification techniques for data abstractions. *IEEE Trans. on Software Engineering 1*, 1 (March 1975). Also in Yeh77a.

The authors discuss the role (for correctness proofs and programming methodology) and importance of the formal specification of data abstractions. Five specification techniques are surveyed and analyzed. A methodology using specification is proposed: start with a concept and specify its properties formally using an unambiguous mathematical language, develop a module for each concept and prove it correct with respect to the specification, use the specification for communication with users and implementors, prove the whole program by proving its component modules.

The authors offer six criteria for evaluating specification methods: formality, constructability, comprehensibility, minimality, wide range of applicability, and extensibility. They discuss the data abstraction as an appropriate unit of specification and the properties of those specifications. They divide such specifications into syntactic and semantic parts. Five specification techniques are evaluated: fixed disciplines (e.g. graphs or sets), arbitrary disciplines, state model (e.g. Parnas), axiomatic descriptions (e.g. Hoare), and algebraic definitions. The authors conclude that no technique is uniformly superior. The methods vary in representational bias, e.g., implicitly defined objects have no such bias, hence, a wide range of applicability. State machines are least extensible. All methods need formalization as well as error and exception handling techniques.

Liskov77a
Liskov, B. H. and Berzens, V. An appraisal of program specifications. in Wegner, P. (Ed.), *Research Directions in Software Technology*, MIT Press, 1979, pp. 276-301.

Liskov77b
Liskov, B. H., Snyder, A., Atkinson, R. and Schaffert, C. Abstraction mechanisms in CLU. *Comm. ACM 20*, 8 (Aug. 1977).

CLU is a new programming language designed to support the use of abstractions in program construction. Work in programming methodology has led to the realization that three kinds of abstractions, procedural, control, and especially data abstractions, are useful in the programming process. Of these, only the procedural abstraction is supported well by conventional languages, through the procedure or subroutine. CLU provides, in addition to procedures, novel linguistic mechanisms that support the use of data and control abstractions. This paper provides an introduction to the abstraction mechanisms in CLU. By means of programming examples, we illustrate the utility of the three kinds of abstractions in program construction and show how CLU programs may be written to use and implement abstractions. The CLU library, which permits incremental program development with complete type-checking performed at compile-time is also discussed. (authors' abstract)

The authors motivate the role of abstraction in the design and construction of programs. They use data abstraction, i.e., the behavior of a class of objects, in a many stage successive refinement or decomposition of a problem into subproblems. Modules are used to implement the resulting abstractions.

Liskov78
Liskov, B. H., Schaffert, C., Scheifler, B. and Snyder, A. *CLU Reference Manual*, Computation Structures Group Memo 161, MIT Lab, July 1978.

Lockemann79a
Lockemann, P. C., Mayr, H. C., Weil, W. H. and Wöhlleber, W. H. Data abstractions for database systems. *ACM TODS 4*, 1 (March 1979).

The authors apply data abstraction techniques to database systems in order to explore the capabilities and shortcomings of existing data definition and manipulation facilities. The axiomatic method for specifying data abstractions is introduced and used to give precise definitions of the following intuitive concepts: a data model is a collection of data type concepts (parameterized data types) which are defined by a set of operations, an axiom schema, and a set of relations; a schema is a collection of data types which result from tailoring data type concepts with particular parameters; and a database is a collection of data type instances. A methodology for database design using these concepts defined in terms of abstraction is given: first the data model is constructed, then data type concepts are selected and specialized to form types appropriate for the application, finally structural and operational constraints are added.

The authors present a specification methodology for data types which uses Guttag-like axioms, relations, and constraints. Examples are given of type specifications for a database application. Partial mappings are treated by explicit enumeration and by predicates. Interdependent events are treated by means of pre-, coincident, and post-conditions.

The paper extends Guttag's specification technique by means of "relations," enumeration, and predicates to handle partial functions and mappings. The paper is an excellent introduction to the application of data abstraction to databases. However, it treates data structure as implicitly defined through operations; structure is treated as representational; the problems of specifying highly interpreted, complex schemas are not addressed. The only data abstraction mechanism discussed is the abstract data type.

Lockemann79b
Lockemann, P. C., Mayr, H. C. and Dittrich, K. R. A pragmatic approach to algebraic specification of software modules. Bericht Nr. 1/79, Fach. Informatik, U. Karlsruhe, Feb. 1979.

An important aspect of modern software engineering is the implementation-independent specification of module interfaces. A number of techniques have recently been proposed for this purpose. Among them, the algebraic specification technique seems to be the easiest to learn and to apply by practitioners. Despite the wealth of literature, little is known of the usefulness of the method for more complex interfaces such as for handling sequential lists, ordered sets, files, or databases.

The paper examines the adequacy of the algebraic technique for some of the interfaces mentioned above. Some of the extensions that have recently been proposed such as parameterized types and hidden operators are shown to extend the applicability of the techniques considerably. A number of notions such as concept, type, index, or structure are more precise and strict rules for formulating the axioms are introduced. Some limitations of the technique are pointed out. The discussions are accompanied by a number of examples to demonstrate the legibility and comprehensiveness of the technique. (from authors' extract)

The authors extend the work described in their TODS paper. The notions are elaborated, extensive examples are given, and new problems (e.g. hidden operations) are addressed. The paper investigates the usefulness of the algebraic specification technique for non-trivial data structures. The authors conclude that although these specifications improve communications between users, systems designers, and programmers, some fundamental extensions are required for database systems, e.g., triggers.

London77
London, R. L. Perspectives on program verification. In Yeh, R. T. (Ed.) *Current Trends in Programming Methodology Vol. II*, Prentice-Hall, Englewood Cliffs, N.J., 1977, pp. 151-172.

London78a
London, R. L., Guttag, J. V., Horning, J. J., Lampson, B. W., Mitchell, J. G., and Popek, G. J. Proof rules for the programming language Euclid. *Acta Informatica 10*, 1-26 (1978).

London78b
London, R. L., Shaw, M., Wulf, W. A. Abstraction and verification in ALPHARD: A symbol table example. In Hibbard, P. G., and Schuman, S. A. (Eds.), *Constructing Quality Software*, North-Holland, 1978, pp. 319-351.

MacEwen78
MacEwen, G.H. and Matin, T.P. The use of abstract data types in top down design. TR 78-70, Queens University, Kingston, Canada.

This paper proposes the use of the abstract data type as the primary unit of decomposition in top-down software design. It is suggested that programming-in-the-large can be accomplished in a language for programming-in-the-small augmented with abstract data types. A complete structural design can thus be produced within the same language framework as wll be used for implementation. An extensive non-trivial example is shown to support the proposal. (authors' abstract)

Maier79
Maier, D., Mendelzon, A. and Sagiv, Y. Testing implications of data dependencies. *ACM TODS 4*, 4(Dec. 1979), pp. 455-469.

Majster77
Majster, M. Limits of the "algebraic" specifications of abstract data types. *SIGPLAN Notices 12*,10 (Oct. 1977).

This short paper deals with the feasibility of the algebraic specification technique for abstract data types. The author discusses some inherent difficulties of the technique including the necessity of an infinite characterizing set of axioms or relations between sequences of operations. A traversal stack is used to demonstrate the inapplicability of the technique.

Majster79a
Majster, M. E. Treatment of partial operations in the algebraic specification technique. In PSRS79.

The author deals with some inherent difficulties in applying the algebraic specification technique to abstract data types. The problems include: (1) the necessity of hidden or auxiliary operations which increase the size of a specification and which require implementation and (2) the problem of deriving the equality of terms when partial operators are expressed through "error" equations. The author proposes means for avoiding problems with partial operations.

Majster79b
Majster, M. E. Data types, abstract data types, and their specification problem. *Theoretical Computer Science 8*, 1 (1979). pp. 89-127.

The author makes the basic point that data types, abstract data types, and the algebraic specification of data types are three different things. She points out that abstract data types play an important role in modularization, the reliability, and the verification of programs, however, there is no consensus as to the definition of the concept of data type. The following definitions are offered: a data type is a set of objects together with a finite set of operations on those objects; an abstract type is a factor structure W/E where W is a word algebra, i.e., the set of all terms built from operation symbols and objects; an algebraic specification consists of a set of equations expressing the properties of opertions. An algebraic specification generally specifies more than one data type.

The paper discusses these three concepts mathematically and raises a number of problems in the specification of abstract data types, e.g., some data types are not finitely presentable using existing specification techniques and large numbers of auxiliary operations are frequently needed. The author proposes that abstract data types should be specified "up to isomorphism" so that users (at user interfaces) are concerned only with how objects behave and not what they look like.

At the VLDB panel session in Berlin (1978) on the use of abstract data types in databases the author made the following remarks. Data types can be used to (i) stress the dynamic aspects of databases, (ii) enforce "legal" operatons, and (iii) incorporate integrity constraints within type definitions. Abstract data types can be used to specify the behavior of user visible objects "up to isomorphism" thereby providing some degree of data independence. The use of algebraic specifications in databases is more complex. Not all data types can be specified. Frequently, a large number of "hidden" or auxiliary operations are needed which complicate an already complex problem. Finally, since the specification technique is based on the interaction of operations, it is not well adapted to changes in the operations.

Manna79
Manna, Z. and Waldringer, R. Synthesis: dreams-->programs. *IEEE Trans. Soft. Eng. SI-5*, 4 (July 1979).

Deductive techniques are presented for deriving programs systematically from given specifications. The specifications express the purpose of the desired program without giving any hint of the algorithm to be employed. The basic aproach is to transform the specifications repeatedly according to certain rules, until a satisfactory program is produced. The rules are guided by a number of strategic controls. These techniques have been incorporated in a running program-synthesis system, called DEDALUS.

The techniques are illustrated with a sequence of examples of increasing complexity; programs are constructed for list processing, numerical calculation, and array computation. The methods of program synthesis can be applied to various aspects of programming methodology, program transformation, data abstraction, program modification, and structured programming.

The DEDALUS system accepts specifications expressed in a high-level language, including set notation, logical quantification, and a rich vocabulary drawn from a variety of subject domains. The system attempts to transform the specifications into a recursive, Lisp-like target program. Over one hundred rules have been implemented, each expressed as a small program in the QLISP language. (from authors' abstract)

Manola77
Manola, F. Abstract data types in data base models and architecture. Computer Corp. of America, Boston.

The author raises a number of questions and speculates on the application of data abstraction to databases. He notes the behavioral emphasis with abstract data types versus the representational emphasis with high-level data models. His main concern is that of type extensibility.

He raises the following issues: abstract data types are a "starting point" from which users construct their own "models" consisting of constructs and operations relevant to the world being modelled. Data models, on the other hand, present a fixed set of objects, relationships, and operations over them. Can these two approaches be resolved? If we view current data models as primitives in a data abstraction facility, the mapping capabilities inherent in abstract data types becomes of more importance than the constructs of the data model.

How does the construction of new concepts from old ones relate to a database architecture? Perhaps data abstraction will permit specialization of general views and concentration on aspects essential to a viewer.

How does programming with abstract data types affect our ideas of "procedurality?" Data models support storage and retrieval but do little for database programming (manipulation).

Can languages supporting data abstraction be used to provide good DBMS interfaces?

McGee76
McGee, W. C. On user criteria for data model evaluation. *ACM TODS 1*, 4 (Dec. 1976).

McLeod76
McLeod, D. J. High level domain definition in a relational database system. In Utah76.

The author applies data abstraction and data type concepts to databases to aid database design and semantic integrity. A relational database is herein defined as a collection of normalized relations (relations in first normal form) and as a collection of domains. A normalized relation may be viewed as a table, wherein each row of the table corresponds to a tuple of the relation, and the entries in a given column belong to the set of values constituting the domain underlying that column. The domains of a database have an abstract existence apart from the database relations.

The database also includes various types of semantic integrity rules, which specify additional properties of the data in the database. One such type of semantic integrity rule is the domain definition. A domain definition includes the precise description of the set of values (objects) constituting the domain. In a normalized database, all domains are sets of atomic data values. A domain definition also includes a specification of the ordering on the values in a domain, for comparability purposes. In addition, a domain definition contains a specification of the action that is to occur if an attempt is made to violate the restriction that every entry in each column of a relation must be from the underlying domain of that column.

A non-procedural language permitting the high level expression of domain definitions is defined. Language details and examples are presented, and the syntax and informal semantics of the domain definition language are given. This approach to domain definition is analyzed in terms of its impact on other aspects of database semantic integrity. The relationship with the database system in general is outline. An analysis of intradomain and interdomain comparability is included. An introduction to relevant implementation issues is also presented. Emphasis is placed on a general approach to implementation and implementation techniques, rather than on a specific system. (author's abstract)

McLeod78a
McLeod, D. A semantic data base model and its associated structured user interface. Ph.D. diss., MIT, August 1978.

The conventional approaches to the structuring of databases provided in contemporary database management systems are in many ways unsatisfactory for modelling database application environments. The features they provide are too low-level, computer-oriented, and representational to allow the semantics of a database to be directly expressed in its structure. The semantic data model (SDM) has been designed as a natural application modelling mechanism that can capture and express the structure of an application environment. The features of the SDM correspond to the principal intensional structures naturally occurring in contemporary database applications. The SDM provides a rich but limited vocabulary of data structure types and primitive operations, striking a balance between semantic expressibility and the control of complexity. Furthermore, facilities for expressing derived (conceptually redundant) information are an essential part of the SDM; derived information is as prominent in the description of an SDM database as is primitive data.

The SDM is designed to enhance the effectiveness and usability of database systems:

1. SDM databases are to a large extent self-documenting, in the sense that the description and structure of a database are expressed in terms which are close to those used by users in describing the application environment.
2. The SDM can support powerful user interface facilities, and can improve the user interface effectiveness for a variety of types of users (with varying needs and abilities). Significantly, SDM databases capture information helpful in new database uses to be defined in the database structure. In particular, the SDM supports an incremental, interactive interface for the "naive" nonprogrammer (an interaction formulation advisor), which guides the user through the database and the process of formulating a query or update request.
3. The SDM can be used as a tool in the database design process. The SDM aids in the identification of relevant information in a database application environment, as well as in organizing that information and relating it to its possible users. This can greatly improve the design of lower-level, conventional databases.

The use of the SDM is not dependent on the successful implementation of a new database management system that directly supports it. There are many database management systems in use today, which represent a considerable investment on the parts of their developers and users; the SDM can be effectively used in conjunction with these existing database systems to enhance their effectiveness and usability. For example, a prototype interaction formulation advisor demonstrates that the SDM can be used as a user-oriented "front end" to a conventional database system; an analysis of application modelling with the SDM illustrates its effectiveness in improving and simplifying the database design process. (author's abstract)

McLeod78b
McLeod, D. A disciplined approach to user-database interaction. Computer Science TR-79-3, University of Southern California, Dec. 1978.

There are significant limitations to conventional approaches of supporting end-users of a database system who are nonprogrammers and naive of the content and structure of a database. There are several keys to effectively providing direct database access for such users: the user interface should be based on a simple database model, whose structures are close to the natural constructs in the application environment; the user should be given interactive assistance and guidance in the process of formulating a database query (or update); the system should aid the user in breaking up a complex query into manageable parts, and the user should be expected to make only simple sorts of decisions during the interaction formulation process; the user's view of a database should be adaptable, so that useful derived information can be captured and made available to simplify future queries. A prototype user tool, called an interaction formulation advisor,

has been designed and implemented based on these principles. This user interface facility is founded on a model of the query formulation process. Our experience with this model and tool has shown that it has many advantages over the very high level language and natural language approaches to facilitating direct database access for naive nonprogrammers. (author's abstract)

Mealy67
Mealy, G. Another look at data.*Proc. AFIPS* Vol. 31(1967).

Mealy77
Mealy, G. Notions. In Yeh77a.

Melkanoff78
Melkanoff, M. A. and Zamfir, M. The axiomatization of .data base conceptual models by abstract data types. UCLA-ENG-7785, Computer Science Department, UCLA, January 1978.

This paper presents abstract data type definitions of the essential data structures of the hierarchical database model. These include: lists, k-lists, and dynamic hierarchies. The definitions are formal, long, and complex; however, the authors argue that this complexity is inherent in the data structures being defined. The authors claim that the remaining features of the hierarchic model may be defined by additional operators and axioms. They claim that the relational and network models can also be so specified. They conclude that abstract data types can be used as a tool for defining data types of arbitrary complexity.

Melliar77
Melliar-Smith, P. M. and Randell, B. Software Reliability: the role of programmed exception handling. In PLDRS77.

The paper discusses the basic concepts underlying the issue of software reliablity, and argues that programmed exception handling is inappropriate for dealing with suspected software errors. Instead it is shown, using an example program, how exception handling can be combined with the recovery block structure. The result is to improve the effectiveness with which problems due to anticipated faulty input data, hardware components, etc., are dealt with while continuing to provide means for recovering from unanticipated faults, including ones due to residual software design errors. (authors' abstract)

Melliar79
Melliar-Smith, P. M. *Tutorial on System-Specifications*, IEEE Computer Society, April 1979.

Merrett77
Merrett, T. Aldat - Augmenting the relational algebra for programmers. TR SOCS-78.1, School of Computer Science, McGill University, Nov. 1977.

Minker75
Minker, J. Performing inferences over relational databases. *ACM SIGMOD Workshop*, San Jose, Calif., 1975.

Minker79
Minker, J. and Zanon, G. Consistency and integrity in databases. TR-723, Computer Science, University of Maryland, 1979.

Minsky75
Minsky, M. A framework for representing knowledge. In Winston, P. (Ed.), *The Psychology of Computer Vision*, McGraw Hill, 1975.

Mitchell76
Mitchell, J. and Wegbreit, B. A next step in data structuring for programming languages. In Utah76.

The authors propose a framework for research into the use of parameterized types to implement data abstractions. Abstractions and the authors' concept of a scheme are compared. An abstraction involves a representation-free specification of properties of a data type. Programs using abstractions can be developed and verified without regard to how abstractions are realized. A scheme is a parameterized model for a set of types. It is a module that takes values and types as parameters. The advantage claimed for schemes is that libraries of completely general data structuring mechanisms could be built. These structures could then be specialized for specific properties without having to program them from scratch. The authors raise several considerations for pursuing research on schemes, e.g., direct manipulation of objects, access control, and the imposition of constraints on parameters.

Mitchell77
Mitchell, J. G. and Wegbriet, B. Schemes: A high level data structuring concept. CSL-77-1. Xerox, PARC, Jan. 1977.

In recent years, programming languages have provided better constructs for data type definitions and have placed increasing reliance on type machinery for protection, modularization, and abstraction. This paper introduces several new constructs which further these ends. Types may be defined as similar to existing types, *extended* by additional properties. *Schemes* are type-parameterized definitions. For example, symbol tables and symbol table operations can be defined as a scheme with the key and value types as parameters; an instantiation of the scheme implements a specific type of symbol table. Because new types are typically defined along with other related types, an instantiated scheme may *export* a set of new types. A set of schemes with a common name and common external behavior can be viewed as alternative implementations of an *abstraction.* Parameter specifications associated with each scheme are used to select the appropriate implementation for each use. (authors' abstract)

Mitchell79
Mitchell, J.G., Maybury, W. and Sweet, R. Mesa Language Manual, Version 5.0. CSL-79-3, Xerox PARC, April 1979.

The Mesa language is one component of a programming system intended for developing and maintaining a wide range of systems and applications programs. Mesa supports the development of systems composed of separate modules with controlled sharing of information among them. The language includes facilities for user-defined data types, strong compile-time checking of both types and interfaces, procedure and coroutine control mechanisms, and control structures for dealing with concurrency and exceptional conditions. (authors' abstract)

Model79
Model, M. L. Monitoring system behavior in a complex computational environment. CSL-79-1, Xerox PARC, Jan. 1979.

Complex programming environments such as the representation systems constructed in AI research present new kinds of difficulties for their users. A major part of program development involves debugging, but in a complex environment, the traditional tools and techniques available for this task are inadequate. Not only do traditional tools address state and process elements at too low a conceptual level, but an Artificial Intelligence system typically imposes its own data and control structures on top of those of its implementation language, thereby evading the reach of traditional program-level debugging tools. This work is directed at the development of appropriate monitoring tools for complex systems, in particular, the representation systems of Artificial Intelligence research.

The first half of this work provides the foundation for the design approach. The nature of computer programs is discussed, and a concept of "computational behavior" defined. A thematic survey of traditional debugging tools is presented, followed by a summary of recent work. Observation of program behavior ("monitoring") is shown to be the main function of most debugging tools and techniques. Particular difficulties involved in monitoring the behavior of programs in large and complex AI systems are discussed.

The second half presents an approach to the design of monitoring facilities for complex systems. A new concept called "meta-monitoring" replaces traditional dumps and traces with selective reporting of high-level information about computations. The importance of the visually-oriented presentation of high-level information and the need to take into account differences between states and active processes are stressed. A generalized method for generating descriptions of system activity is developed. Some specific display-based monitoring tools and techniques are exhibited. Several of the experimental monitoring facilities are described and their application to existing Artificial Intelligence Systems illustrated. (from author's abstract)

Morris79
Morris, J.B. Data abstraction: A static implementation strategy. *SIGPLAN Notices 14*, 8 (1979).

A description of the implementation of the data abstraction mechanism for the Model programming language is discussed. The Model programming language utilizes a "static" approach to data abstraction in that abstract concepts are expanded by early phases of the compiler in a controlled fashion into base language concepts. Thus, the final phase of the compiler sees only base language constructions. Exemplified are generic procedures accepting parameters of several different data types. A discussion is given of the problem of "parameter explosion," a problem traditionally inherent in languages implementing open (inline) procedures. (author's abstract)

Morris73
Morris, J. H. Types are not sets. *Proc. ACM POPL*, Oct. 1973. pp. 120-124.

The author uses the notion that a type is not a subset of a universe of values, as in mathematics, to discuss the role of type checking. A type system, through its type checking facilities, constrains program interaction to provide, for a given type, authentication (only values of that type (representation) can be submitted for processing by given programs) and secrecy (only the given procedures can be applied to objects of that type). It is argued that objects of a given type be created, manipulated, etc. by the owner (definer) of the type through modules which provide users with type names and operations. Modules have write access to its local values and may selectively give out read access to such values. The author discusses proof rules for verifying module usage and relates his module concept to those of Simula classes and polymorphic functions.

Mylopoulos78
Mylopoulos, J., Bernstein, P. A. and Wong H. K. T. A preliminary specification of TAXIS: A language for designing information systems. Tech. report CCA-78-02 (Jan. 1978).

Abstraction mechanisms from artificial intelligence, database management and programming languages are incorporated into a language framework for the design of interactive information systems. In particular, the technologies developed for semantic networks, relational databases and abstract data types are integrated into a language called TAXIS.

The basic concepts in TAXIS are those of *class*, *property* and the *ISA* (generalization) *relationship* between classes. Through these concepts a framework is developed that offers database management facilities, a means of specifying integrity constraints, exception handling, and organizational principles for TAXIS programs. The report includes a detailed description of language constructs as well as a long example of a TAXIS program that models aspects of a University Students' world. (authors' abstract)

A novel feature proposed here for programming languages is a generalization hierarchy of procedures and the ability to specialize a procedure by means of constraints and additional properties in particular contexts. TAXIS combines both procedural and structural descriptions of data abstractions in what is called a class.

Mylopoulos79
Mylopoulos, J., Bernstein, Pl, Wong, H. K. T. A language facility for designing interactive database intensive applications. CSRG TR-105, Computer Systems Research Group, University of Toronto, 1979.

Nordstrom78
Nordstrom, B. Assignment and high level data types. *Proc. 78 Annual ACM Conf.*, Dec. 1978, pp. 630-638.

Norman75
Norman, D. A. and Rumelhart, D. E. (Eds.). *Explorations in cognition*, San Francisco, Freeman, 1975.

NYU78
Proc. NYU Symposium on Database Design. New York University, New York, May 1978.

Palmer78
Palmer, I. Practicalities in applying a formal methodology to data analysis. In NYU78.

Paolini79a
Paolini, P. *Abstract data types for database management systems architecture.* Ph.D. diss. In progress, DCS, UCLA.

The author investigates the uses of abstract data types to construct external views and to map them into a common conceptual view as described in the ANSI/X3/SPARC framework. The author claims that the following steps are both necessary and possible:
1. specify external views in terms of structures and operations,
2. develop languages to implement external views by translating external transactions into operations at the conceptual level,
3. formally define and prove the correctness of view implementation,

4. formally define and prove the correctness of view interference (cooperation), and
5. develop tools for the definition and proof of application programs.

These steps require some extensions to abstract data type techniques, chiefly, abstractions which share part of a common representation and predicate schemes which acknowledge interference or cooperation of many different views.

Paolini79b
Paolini, P. Verification of views and application programs. IEE Tech. Report Politechnico di Milano, Oct. 1979.

Formal specifications are needed to verify the correctness of application programs. Therefore, formal specifications must be given for views on which these programs operate. This paper proposes a method for the formal definition of views. A view can be described as a set of abstract objects; each one of these objects belongs to a given abstract data type. These objects can be manipulated through the operations (transactions) allowed for that view.

Using the proposed method, proofs of correctness of application programs can be carried out. New problems arise when several views share a common representation. In particular, the abstract definition of a view can be inadequate for the description of the effects on a view which is derived from the execution of operations on other views. A solution for this problem is proposed, its usefulness and its limits are discussed.

Parnas70
Parnas, D. L. A technique for software module specification with examples. *Comm. ACM 15*, 5 (1970).

The author presents an approach for writing software specifications which adheres to the principle of non-interference (Dijkstra). The principle of information hiding aims at providing users with only the information needed to code it. Specifications should be formal enough to be machine checkable and be understandable by users and implementors. It is proposed that specifications be checked for completeness and correctness before coding. The method, called the abstract machine approach, is only at a very early stage of development and use. A module (its function) is specified by giving its set of possible values, initial values, parameters and the effects of the function (almost all information goes here).

Parnas72
Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, (Dec. 1972).

The desirability of breaking programs into separate modules is well known and generally accepted, however, hitherto, little has been written on the rules for determining the module boundries.

The author contrasts two different methods of dividing a KWIC Index production system into modules. The decomposition that is based on the major steps as seen in a flowchart view of the process is shown to be much less amenable to subsequent design changes than one that is founded on the author's criterion of "information hiding" where each module contains some piece of the design that is known only to itself. The module interface specifications contain no more than the minimum of information required by the caller and the implementor. The less these two know about each other's module the better, since this allows the implementor freedom to make later improvements without affecting the caller, and also permits a wider variety of potential callers thus extending the usefulness of the module. As a consequence, for example, the details of a data structure and its access mechanism should be kept within one module at the lowest possible level. This principle leads to a hierarchical structure that corresponds closely to Dijkstra's

concept of levels of abstraction.

One problem with the author's preferred method of decomposition is an apparent loss of understandability of the process; however, I think this is a question of presentation rather than it being inherent. Another problem with this method is that, since the rules are not based on the order in time in which processing is expected to take place, there is the possibility for loss of efficiency due to heavy linkage overhead. The author touches on but does not resolve this problem.

Apart from the point noted, the paper is clearly written and is recommended, not only to those who design systems but also to those who implement the modules, since its precepts should be applied at all levels. (CR25, L. A. Hollaar)

Parnas75
Parnas, D. L. The influence of software structure on reliability. In Yeh77a and in PSE75.

The author argues that reliability and correctness are not synonymous. Reliability is a statistical measure referring to the demands placed on the system. Correctness refers to the extent to which a system meets its specifications. Hence, reliable systems need not be correct and *vice versa*. It is argued that although reliability can be improved through concern for software structure, correctness is currently beyond our reach. Structure is used to mean the way in which a software system is divided into modules, the nature of module interfaces, and the assumptions various modules make about each other.

The author offers the following advice in designing software:
1. specify the desired behaviour of modules, including that desired in error conditions,
2. specify interfaces both for normal and abnormal situations, and
3. include conventions for communicating error conditions between modules.

An appendix illustrates the points in the paper through examples and questions to be considered when designing and defining interfaces.

Parnas76
Parnas, D. L., Shore, J. E. and Weiss, D. Abstract data types defined as classes of variables. In Utah76.

The authors discuss the nature of the data type concept. Five previous approaches to a definition of a type are characterized as: syntactic - type is the information one gives a variable in a declaration; value spaces - a type is a set of values; behavior - a type is a value space and a set of operations on elements in the space; representation - a type is determined by its composition of primitive types; representation plus behavior - a type is defined by a representation plus a set of operations.

They outline the motivations for type extension: abstraction (a concept with more than one realization), redundancy and compile time checking, abbreviation of code, and data portability. The five previous approaches are inadequate or too restrictive to meet these goals.

The authors propose a new approach in which types are defined as an equivalence class of variables and variables are considered primitive. The authors define *mode* as a class of variables for which any one can be legally substituted for any other. An abstract data type is a collection of modes. The collections are constructed for a number of reasons: set of modes satisfying the same specification but requiring recompilation or substitution, modes with the same representation but different operator effects, modes that are invocations of a parameterized mode description (for code sharing), and modes which share common properties but differ in others. One claim for modes is that a variable may be of more than one type. This requires a general equivalence facility for variables. Many of these concepts relate strongly to type generalization.

Parnas77
Parnas, D. L. The use of precise specifications in the development of software. *Proc. IFIP*, 1977.

The author discusses the role of formal and precise specifications in the development of "correct" software. The author asserts that the specification and verification of intermediate design decisions are essential for the development of large software products but that formal specifications provide only one approach. The author distinguishes between engineering specifications (a precise statement of requirements a product must satisfy) and an abstract program (one in which operations and data elements are used but not implemented) which represents a class of programs.

Specifications are needed: to describe the problem, as a means of communication, to permit programmers to concentrate on components essential to their task, to aid intermediate design decisions, and to support verification. Specifications must be precise, abstract (i.e., free from implementation details) and unambiguous (e.g., natural language is therefore dangerous).

The author proposes a "mixed" methodology using information hiding (IH) module specifications which involves (a) the interface definition for an IH module and (b) a completely documented abstract program. He proposes the use of predicate transformers and abstract specifications and gives examples.

The author concludes by emphasizing the importance of specifications to record design decisions, to guide problem decomposition, to define "correctness". However, he points out that no complete verification techniques exist.

Parnas79
Parnas, D. L. Designing software for ease of extension and contraction. *IEEE Trans. on Soft. Eng. SE-5*, 2 (March 1979).

The author addresses the goal of designing for change, i.e., the reduction or extension of a system's capabilities. The benefits include: the potential of design, correction, implementation, and testing throughout systems development. Problems to overcome are: information is poorly and redundantly distributed in systems; programs are often part of fixed chains of control and can not be removed; programs perform more than one function; and interdependencies due to loops in the "uses" hierarchy.

The author proposes steps to improved systems structure and illustrates them with an example.
1. Define minimal requirements of programs before the design stage then consider only minimal increments of capability.
2. Information hiding (encapsulation, abstraction): identify changables as "secrets," isolate specialized components, and define interfaces to insulate change.
3. Virtual machine concept: design software as an analog of hardware components and their instructions.
4. Design the "uses" structure: A "uses" B is the correct operation of A requires the correct operation of B. There must be no loops. A must be simpler because it uses B and B must not be more complex because it does not use A. A and B must be independently part of useful subsets of the system.

The author notes that the following pattern facilitates design. Low level operations assume fixed data structures. The next level permits swapping of similar data elements. High level programs allow the creation and deletion of such data elements.

Pirotte78
Pirotte, A. Linguistic aspects of high-level relational languages. MBLE Report R367, January 1978.

Popek77
Popek, G. J., Horning, J. J., Lampson, B. W., Mitchell, J. G., and London, R. L. Notes on the design of EUCLID. In PLDRS77.

This paper discusses a number of issues in the design of Euclid including such topics as the scope of names, aliasing, modules, type checking, and the confinement of machine dependencies. The authors state why they believe that programming in Euclid will be more reliable than programming in PASCAL.

Several points are made about modules which are used for data encapsulation (the packaging of data structure and manipulating operations) and information hiding. A Euclid module is like a record with the routines, types, initialization and finalization mechanisms and an export list to control the external visibility of identifiers. A module is to be used as a type constructor to create many instances of a type and to express iteration over the abstract data types they define.

A contribution to the notion of type is the concept of "same type." "A type is the same as its definition." This resolves some problems of type checking while maintaining a strongly typed language.

Prenner77
Prenner, C. A uniform notation for expressing queries. ELR M77/60, Electronics Research Laboratory, U. C. Berkeley, Sept. 1977.

Prenner78
Prenner, C. J. and Rowe, L. A. Programming languages for relational database systems. *Proc. National Computer Conf.* 1978.

The authors describe the lack of interaction between programming language and relational database systems. They propose an integrated approach to providing high level programming language access to databases.

Programming language access to databases is either through subroutine calls which are cryptic and inflexible or through a preprocessor which involves type conversions, awkward programming style, and a loss of type checking strength. Typically database languages are less procedural than programming languages. Database and programming language type systems are distinct and require complex conversions or the restricted use of both type systems. Data abstraction has been proposed as a basis of integrating the two areas but few results have yet emerged.

The authors propose to extend the type system of a general high order language so as to accommodate database access directly. Examples are given to illustrate the proposal. Benefits claimed for such an approach are: increased modularity and data integrity, no type conversions, possible type extensibility, choice of binding times, and optimization potential.

The authors propose that data abstraction may offer solutions for such database issues as views, integrity constraints, and triggers.

PSCC79
Proc. SIGPLAN Symposium on Compiler Construction. SIGPLAN Notices 14, 6 (August 1979).

PSE75
Proc. 1975 Int'l. Conf. on Reliable Software. SIGPLAN Notices 10, 6 (June 1975).

PSE76
Proc. 2nd Int'l. Conf. on Software Engineering, San Francisco, CA, October 1976.

PSE78
Proc. 3rd Int'l. Conf. on Software Engineering, Atlanta, Georgia, May 1978.

PLDRS77.
Wortman, D. B. (Ed.) *Proc. An ACM Conf. on Language Design for Reliable Software*, in *SIGPLAN Notices 12*, 3 (March 1977).

PSRS79
Proc. Specifications of Reliable Software, IEEE Computer Society, May 2-4, 1979.

Quillian68
Quillian, M. R. Semantic memory. in *Semantic Information Processing*, M. Minsky (Ed.), MIT Press, 1968.

Ramamoorthy78a
Ramamoorthy, C. V. and So, H. H. Software requirements and specifications: Status and perspectives. In Ramamoorthy78b.

This paper presents a comprehensive survey of the techniques, tools, and methodologies being used and developed for software specification. The authors present these techniques within a framework based on the software development life cycle. The techniques range from the precise, formal specification of properties of small program modules and data abstractions to specification of structural and flow aspects of whole systems. The paper gives a rather broad perspective of the area.

Ramamoorthy78b
Ramamoorthy, C. V. and Yeh, R. T. *Tutorial: Software methodolgy*, IEEE Computer Society, 1978.

The tutorial collects together some of the most important papers on software methodologies. The tutorial is divided into five sections: requirements formulation, modelling, design, implementation, and case studies. Each section is preceded by an introduction which relates the papers included in the section and outlines the concepts and techniques related to the subject matter of the section.

Randell78
Randell, B., Lee, P. A. and Treleaven, P. C. Reliability issues in computing system design. *ACM Computing Surveys 10*, 2 (June 1978).

REDL78
Informal language specification (REDL), submitted to the U.S. Department of Defense, Feb. 1978.

Reiter76
Reiter, R. Query optimization for question-answering systems. *Proc. of the COLING Conf.*, Ottawa, June 1976.

Reynolds75
Reynolds, J. C. User-defined types and procedural data structures as complementary approaches to data abstraction. *Proc. Conf. on New Directions in Algorithmic Languages*, IFIP, Munich, Aug. 1975.

Riddle78
Riddle, W. E., Wileden, J. C., Sagler, J. H., Segal, A. R., and Stavely, A. M. Behavior modelling during software design. *IEEE Trans. on Soft. Eng.*, SE-4, 4 (July 1978), pp. 283-292.

This paper presents a formalism and language for software design. The modelling scheme presents a medium for the rigorous, formal, and abstract specification of large-scale software system components. It allows the description of component behavior without revealing the internal details. Both collections of sequential processes and the data objects they share may be described. It is argued that the scheme is of particular use during the earlier stages of software design, when the system's modules are being delineated and their interactions designed, and when rigorous, well-defined specifications of undesigned components allow formal and informal arguments concerning the design's correctness. (from author's abstract)

Rieger76
Rieger, C. An organization of knowledge for problem solving and language comprehension. *Artificial Intelligence*, 1976, pp. 86-127.

Roberts77
Roberts, R. and Goldstein, I. The FRL priner. MIT AI memo 408, 1977.

Robinson77
Robinson, L. and Levitt, K. N. Proof techniques for hierarchically structured programs. *Comm. ACM 20*, 4 (April 1977).

A method for describing and structuring programs that simplifies proofs of their correctness is presented. The method formally represents a program in terms of levels of abstraction, each level of which can be described by a self-contained nonprocedural specification. The proofs, like the programs, are structured by levels. Although only manual proofs are described in the paper, the method is also applicable to semi-automatic and automatic proofs. Preliminary results are encouraging, indicating that the method can be applied to large programs, such as operating systems. (authors' abstract)

Rolland79
Rolland, C., Leifert, S., and Richard, C. Tools for information system dynamics management. In VLDB79.

In the former VLDB Conference we have presented a global model for the design of a complete information system (IS). This model constitutes an original attempt to integrate at the conceptual level, data, processes and dynamics in the design of IS. In the same way, we now present a system of tools allowing IS definition, creating and functioning compatible with the general model previously defined. This system is an extension of the DBMS approach for the design and management of IS. The goal of those tools is to create an IS functioning and evolving as the dynamic reality it represents. (authors' abstract)

Ross76
Ross, D. T. Towards foundations of the understanding of types. In Utah76.

Ross77
Ross, D. T. Structured Analysis (SA): A language for communicating ideas. *IEEE Trans. on Software Engineering 3*, 1 (Jan. 1977).

Roubine76
Roubine, O. and Robinson, L. SPECIAL (SPECIfication and Assertion Language): Reference manual. SRI International Memo, Aug. 1976.

Roussopoulos75
Roussopoulos, N. and Mylopoulos, J. Using semantic networks for database management. In VLDB75.

Roussopoulos76
Roussopoulos, N. A semantic network database model. Ph.D. thesis, Dept. of Computer Science, Univ. of Toronto, 1976.

Rowe79
Rowe, L. A. and Shoens, K. A. Data abstractions, views and updates in RIGEL. In SIGMOD79.

Language constructs to support the development of database applications provided in the programming language RIGEL are described. First, the language type system includes relations, views, and tuples as built-in types. Tuple-values are introduced to provide more flexibility in writing procedures that update relations and views.

Second, an expression that produces sequences of values, called a generator, is defined which integrates relational query expressions with other iteration constructs found in general-purpose programming languages. As a result, relational expressions can be used in new contexts (e.g., parameters to procedures) to provide new capabilities (e.g., programmer defined aggregate functions).

Lastly, a data abstraction facility, unlike those proposed for other database programming languages, is described. It provides a better notation to specify the interface between a program and a database and to support the disciplined use of views.

All of these constructs are integrated into a sophisticated programming environment to enhance the development of well structured programs. (authors' abstract)

The authors argue that modules are more appropriate for data abstraction in databases than are abstract data types. They state that data abstraction is used to define external schemas which are complex; involve several types, variables, and procedures; and which are instantiated only once. It is argued that

abstract data types are less convenient, although usable, since they are intended for defining new data types which are to have multiple instantiations. The RIGEL has three sections: a public section defining objects and operations on those objects that are accessible to users; a private section which implements those objects and operations; and a section for initializing the module.

Sandewall70
Sandewall, E. Representing natural-language information in predicate calculus. *Machine Intelligence*, 6 (1970).

Scheuermann79
Scheuermann, P., Schiffner, G., and Weber, H. Abstraction capabilities and invarient properties modelling with the entity-relationship approach. *Proc. Int'l. Conf. on E-R Approach to Systems Analysis and Design*, Los Angeles, Dec. 1979.

Schmid75
Schmid, H. A. and Swenson, J. R. On the semantics of the relational model. In SIGMOD75.

Schmidt77
Schmidt, J. W. Some high level language constructs for data of type relation. *ACM TODS 2*, 3 (Sept. 1977).

For the extension of high level languages by data types of mode relation, three language constructs are proposed and discussed: a repetition statement controlled by relations, predicates as a generalization of Boolean expressions, and a constructor for relations using predicates. The language constructs are developed step by step starting with a set of elementary operations on relations. They are designed to fit into PASCAL without introducing too may additional concepts. (author's abstract)

The paper presents an extension of PASCAL called PASCAL/R which was designed to provide high-level programming language access to relational databases with considerable attention to an economy of concepts and to the design philosophy of PASCAL. N. Wirth is very pleased with the extension and has a copy of the PASCAL/R compiler at ETH. A PASCAL/R compiler has been completely operational at the University of Hamburg since late 1977. A new version of the language now exists with relation valued procedure parameters, a more powerful constructor, and the ability to maintain relations within a program but not available to other programs (e.g. local relations). A fully operational compiler with these extensions exists and a number of reasonably large database applications have been developed.

Schmidt78
Schmidt, J. W. Type concepts for database definition. In Shneiderman, B. (Ed.) *Databases: Improving Usability and Responsiveness*, Academic Press, New York, 1978.

Several facilities for type definition are outlined and applied to database definition. A declarative method given by the data structure *relation* is compared with a procedural method given by the capsule mechanism *class*. Comparison is done mainly with respect to type representation and constraint definition. The mapping of relational databases to user programs by means of classes is sketched by an example. (author's abstract)

Schmidt79
Schmidt, J.W. Parallel processing of relations: A single assignment approach. In VLDB79.

PASCAL/R, a language extension based on a data structure relation and some high level language constructs for relations is augmented by a procedure concept for concurrent execution of database actions. Relation type procedure parameters serve two purposes: data accessing and access scheduling. Scheduling requirements are analyzed within the framework of the single-assignment approach and proposals for the stepwise reduction of implementation effort are discussed. (author's abstract)

In this single assignment approach a relation constructor and procedures with relation valued parameters are used to isolate that part of a relltion that is to be altered. Since this can be done in one statement the assignment or modification can be made to appear atomic. These atomic actions can be analyzed for concurrent processing of relations. This is then a high level language solution to the problem of parallel processing of relations.

Schubert76
Schubert, L. Extending the expressive power of semantic networks. *Artificial Intelligence*, 7 2 (Summer 1976).

Shapiro79
Shapiro, S. The SNePS semantic network processing system. In Findler79.

Shaw76
Shaw, M. Research directions in abstract data structures. In Utah76.

The author discusses a number of problems that arise from current attempts to incorporate abstraction mechanisms in programming languages. Some problems are central issues in current research projects; others are direct extensions of current work. Among the problems discussed are: adding abstract data structures to a base language, verification of data abstractions, programmer control over abstractions, parameterization of abstractions, the specification of abstractions. The author speculates on some problems for the future including: multiple implementations for an abstraction, libraries of representation, selection and conversion of representations before and during run time.

Shaw77
Shaw, M., Wulf, W. A. and London, R.L. Abstraction and verification in ALPHARD: Defining and specifying iterators and generators. *Comm. ACM 20*, 8 (Aug. 1977).

The Alphard "form" provides the programmer wth a great deal of control over the implementation of abstract data types. In this paper the abstraction techniques are extended from simple data representation and function definition to the iteration statement, the most important point of interaction between data and the control structure of the language itself. A means of specializing Alphard's loops to operate on abstract entities without explicit dependence on the representation of those entities is introduced. Specification and verification techniques that allow the properties of the generators for such iterations to be expressed in the form of proof rules are developed. Results are obtained that for common special cases of these loops are essentially identical to the corresponding constructs in other languages. A means of showing that a generator will terminate is also provided. (authors' abstract)

Shaw78
Shaw, M., Feldman, G., Fitzgerald, R., Hilfinger, P., Kemurd, I., London, R., Rosenberg, J., and Wulf, W. A. Validating the utility of abstraction techiques. *Proc. 1978 ACM Annual Conf.*, Washington, D. C., Dec. 1978 .

The authors propose that the next significant step in the area of data abstraction is the validation of the underlying hypothesis. The hypothesis is that data abstraction is supported by: 1) data abstraction (encapsulation) mechanism in a language, 2) formal specifictions of abstract properties, 3) verification techniques for abstractions, and 4) a methodology for organizing and developing programs using data abstractions *and* that these facilities lead to significant improvements in program quality.

In actual experience, few programs have been specified and developed using related modularity concepts and no non-toy programs have been developed and verified. The largest programs include: a symbol table (150 lines), message switching (200 lines) CLU compiler (27 ADT's), and network communication with concurrency (1000 lines).

The authors propose controlled experiments to test the hypothesis by considering: feasibility, completeness, quality, life cycle costs, reuseability, and language implementation. The ALPHARD group is currently involved in such a project.

Shipman80
Shipman, D. The functional data model and the data language DAPLEX. To appear *ACM TODS*.

This paper discusses the advantages of a functional approach to data modelling. This approach forms the basis of DAPLEX, a high-level language for data description and manipulation. The paper describes the DAPLEX language and the functional data model and examines the benefits they provide as tools for expressing and solving problems in database management. (author's abstract)

The primary goal of DAPLEX is to provide a formal language for the definition and manipulation of databases which is "conceptually natural", i.e., provides a representation which is close to what a human being would employ. The basic constructs of the language are the entity and the function used to relate entities and model properties of entities. Functions provide multiple user views of the data which are independent of any underlying representation, e.g., tables, however, users can view databases as directed graphs of entities connected by edges defined by functions. Functions may have zero or more arguments as well as inverses. DAPLEX supports sub-types and super-types, derived data, and arithmetic aggregation.

The author claims that DAPLEX was designed to be integrated with a general purpose programming language. Although this is an admirable goal, it is not clear that such an integration would be simple. DAPLEX avoids the update problem of applicative languages by providing a LET statement which is not as "natural" as the rest of the language. Two attractive features of DAPLEX are its use to express constraints and the treatment of meta-data as if it were data.

Shoch79
Shoch, J.F. An overview of the programming language Smalltalk-72. *SIGPLAN Notices 14*, 9 Sept. 1979.

Shneiderman78
Shneiderman, B. Improving the human factors aspects of database interactions. *ACM TODS 3*, 4 (Dec. 1978).

The widespread dissemination of computer and information systems to nontechnically trained individuals requires a new approach to the design and development of database interfaces. This paper provides the motivational background for controlled psychological experimentation in exploring the person/machine interface. Frameworks for the reductionist approach are given, research methods discussed, research issues presented, and a small experiment is offered as an example of what can be accomplished. This experiment is a comparison of natural and atificial language query facilities. Although subjects posed approximately equal numbers of valid queries with either facility, natural language users made significantly more invalid queries which could not be answered from the database that was described. (author's abstract)

Shopiro78
Shopiro, J. E. Theseus - A programming language for relational databases, TR31, DCS, U. Rochester, May 1978.

Theseus, a very high level programing language extending EUCLID, is described. Data objects in Theseus include relations and n-tuples. Primary design goals of Theseus are to facilitate the writing of well-structured programs for database applications and to serve as a vehicle for research in automatic program optimizations. (author's abstract)

SIGMOD75
Proc. ACM-SIGMOD 1975 Int'l. Conference on Management of Data, San Jose, May 1975. Available from ACM, New York.

SIGMOD76
Proc. ACM-SIGMOD 1976 Int'l. Conference on Management of Data, Washington D.C., June 1976. Available from ACM, New York.

SIGMOD77
Proc. ACM-SIGMOD 1977 Int'l. Conference on Management of Data, Toronto, Canada, August 1977. Available from ACM, New York.

SIGMOD78
Proc. ACM-SIGMOD 1978 Int'l. Conference on Management of Data, Austin, Texas, May 1978. Available from ACM, New York.

SIGMOD79
Proc. ACM-SIGMOD 1979 Int'l. Conference on Management of Data, Boston, Mass., May 1979. Available from ACM, New York.

Simmons73
Simmons, R. F. Semantic networks: their computation and use for understanding English sentences. *Computer models of thought and language*, R. C. Schank and K. M. Colby (Eds.), San Francisco, Calif., 1973.

Smith77a
Smith, J. M. and Smith D. C. P. Database abstractions: Aggregation and generalization. *ACM TODS 2*, 2 (June 1977).

The authors describe two forms of abstraction that are fundamental to the design and use of databases: aggregation, which is used to consider a relationship between objects as an aggregate object and generalization, which is used to consider a class of objects as a generic object. They argue that although aggregation is used extensively in programming languages and databases, generalization has been considered almost exclusively in artificial intelligence.

A new data type is introduced as a primitive for defining all objects whether individual, aggregate, or generic in a uniform way. Objects may be defined as aggregation and generalization hierarchies. This structuring discipline is imposed on Codd's relational schema so as to provide a basis for: integrating and maintaining views; data independence; easier understanding of complex systems; natural query formulation; systematic database design; and optimization of lower level implementations. A detailed database application is used to demonstrate these concepts.

The authors present a set of relational invariants which define the properties of the generic type. These invariants must be obeyed by relations if the abstractions are to be preserved. A triggering mechanism for the automatic maintenance of these invariants during update is proposed. The authors argue that generalization and aggregation can be supported using owner-coupled sets of DBTG. (from the author's abstract)

This is one of the most significant contributions to the cross-fertilization of the programming language, database, and artificial intelligence areas. Aggregation and generalization are now widely accepted as fundamental data modelling concepts.

Smith77b
Smith, J. M. and Smith, D. C. P. Integrated specifications for abstract systems. UUCS-77-112, Department of Computer Science, University of Utah, Sept. 1977.

Structural specifications define an abstract object as a composition of other abstract objects. Behavioral specifications define an abstract object in terms of its associated operations. Integrated specifications are a combination of structural and behavioral specifications which are more powerful than either used alone. By providing four naming mechanisms, integrated specifications hide the details of how objects are represented and accessed on storage devices. The four naming mechanisms allow objects to be named in terms of the operations previously applied to them, the unique attributes they possess, the relationships they participate in, and the categories they belong to. Integrated specifications can specify the structure of more abstract systems than the relational database model, while also characterizing dynamic properties. Examples are given of integrated specifications for queue, symboltable, and expression. These specifications are simple and guide, but do not constrain, the implementor in designing refinements. By exploiting abstract structure in specifications, common aspects of inter-object communication can be suppressed and only salient differences emphasized. Integrated specifications can make a significant contribution to the usability, reliability and efficiency of computer systems. (authors' abstract)

Smith78a
Smith, J. M. Comments on papers "A software engineering view of database management" by A.I. Wasserman and "A software engineering view of database systems" by H. Weber. In VLDB78.

The author argues that there is a fundamental incongruity in the treatment of data types between the databases and software engineering (SE) areas. This view is used to criticize the importation into SE of the relation data type and the exportation from SE to the database area of abstract data types (ADT). The

structure of an object concerns its composition from other objects and is orthogonal to its representation which concerns the mapping of the object's structure into storage.

Abstraction provides a means of overcoming the complexity of both the structure and representation of database objects. There are three types of structural abstraction: classification (considering collections of objects as a class or type of object), generalization (considering collections of objects or categories as a higher order class), and aggregation (considering a relationship amongst objects as an aggregate object) through which users can ignore certain application details. Realization is the only form of representation abstraction; implementors use it to hide storage details.

Adding the relation data type to a PASCAL-like language poses the dilemma of violating either the PASCAL philosophy or data independence. PASCAL data types do not support data independence since they are not effective in revealing structure and they provide fixed realizations although they do hide representation detail.

It is argued that ADT's are representational tools and that using ADT's, structural abstractions are treated as realizations. This confusion has serious consequences. Although a change in an object's representation should not alter its operation set, a modification of its structure may require a different set of operations (since the logic may have altered). Another incongruity is that the "uses" structure is usually hierarchic in the representational plane but seldom hierarchic in the structural plane.

The author proposes "abstract syntax" as a basis for structural abstraction and abstract data types as a basis for realization. He claims that one basis for database data types may be an appropriate combination of abstract syntax and abstract data types.

Smith78b
Smith, J. M. A normal form for abstract syntax. In VLDB78.

McCarthy's abstract syntax is the most widely used metalanguage for specifying data structure. It is embedded in various forms in most recent programming languages and data models. A simpler, yet more powerful, abstract syntax is defined which is particularly effective in database applications. An abstract syntax specification shows how objects are composed as the union and cartesian product of other objects. If a specification is not properly constructed, it is demonstrably difficult to write application programs, maintain database integrity and provide graceful evolution. A normal form, called (3,3)NF, for abstract syntax specifications is introduced. Specifications in this normal form are subject to far fewer of the above utilization problems. Unlike previous normal forms which only prescribe composition with respect to cartesian product, (3,3)NF also prescribes composition with respect to union. Examples of normalization are given and the advantages and limitations of the approach are discussed. (author's abstract)

Smith78c
Smith, J. M. and Smith, D. C. P. Principles of database conceptual design. In NYU78.

Criteria and methodologies for the conceptual design of databases, particularly in large and sophisticated applications, are addressed. For a design to be understandable to user and designer alike, intuitive methods for abstracting concepts from a mass of detail must be employed. Two abstraction methods are particlarly important - *aggregation* and *generalization*. Aggregation forms a concept by abstracting a *relationship* between other concepts (called components). Generalization forms a concept by abstracting a *class* of other concepts (called categories). The principle of "object relativity" is essential for the successful integration of abstractions. This principle states that individuals, categories, relationships and components are just different ways of viewing the *same* abstract objects. Using this principle a design may be hierarchically organized into independently meaningful abstractions. An "abstract syntax" is introduced to specify these abstraction hierarchies. An advantage of this abstract syntax is that some concepts do not

have to be arbitrarily classified as "roles". The principle of "individual preservation" is a minimal requirement for maintaining the semantics of aggregation and generalization. It states that every user-invokeable operation must preserve the integrity of individuals. A methodology for designing an abstract syntax specification is outlined. The simplicity of this methodology is directly due to the principles of object relativity and individual preservation. (authors' abstract)

This paper discusses the role of abstraction in the design and specification of database applications.

Smith79
Smith, J. M. and Smith, D. C. P. A database approach to software specification. Computer Corp. of America, CCA-79-17, (April 1979).

A database is a *simulation* of some real-world phenomena that is *represented* on a computing machine. A long-standing goal of database technology has been to specify the semantics of the simulation while suppressing all details of its computer representation. To meet this goal, a variety of specification languages called *database models* has been developed. These models are oriented towards applications where large quantities of intricately inter-related data is manipulated by multiple users.

These applications are, in some ways, more exacting than the systems programming applications which have motivated most other specification languages. A rich typing structure, which includes higher-order types, subtypes and component types, is necessary. A flexible attribute structure, which allows types to have attributes, must also be provided. Data sharing and side-effects must be handled in a simple and natural way. This requires a powerful predicate language for identifying individuals on the basis of their structure. A database approach can therefore bring a different perspective to software specification. (authors' abstract)

This paper presents a new, database approach to software specification. In doing so, the authors survey existing data model concepts and propose a new definition of a data model which includes 1) a data space, 2) a type definition facility, 3) manipulation primitives, 4) a predicate language, and 5) control structures. The concept of subtype is emphasized and the authors' semantic hierarchy model is described. A special notation based on McCarthy's abstract syntax is introduced for type definitions. It is shown how these tools permit the specification of software composed of both conventional structures and databases. The authors claim that this approach permits a better separation between structure (an abstract concept) and representation.

Solvberg79
Solvberg, A. Software requirement definition and data models. In VLDB79.

It is argued that the conceptual schema should contain an ontological subschema (i.e., a "reality" model). It is shown how one particular ontological meta-model can be interfaced to meta-models for the development of software requirement specifications. Through an example it is indicated how the ontological model can provide a basis for proving semantical equivalence/difference of databases. (author's abstract)

Spitzen75
Spitzen, J. and Wegbreit, B. The verification and synthesis of data structures. *ACTA Informatica 4*, pp. 127-144 (1975).

Sptizen78
Spitzen, J. M., Levitt, K. N., and Robinson, L. An example of hierarchical design and proof. *Comm. ACM 21*, 12 (Dec. 78).

Hierarchical programming is being increasingly recognized as helpful in the construction of large programs. Users of hierarchical techniques claim or predict substantial increases in productivity and in the reliability of the programs produced. In this paper we describe a formal method for hierarchical program specification, implementation, and proof. We apply this method to a significant list processing problem and also discuss a number of extensions to current programming languages that ease hierarchical program design and proof.

Standish77
Standish, T. A. Data structures - An axiomatic approach. In Yeh77a.

The author describes the axiomatic approach for defining the properties of data structures. The paper (written in 1973) was perhaps the first paper to discuss data abstraction or data independence viewed as the ability to consider properties of data types independently of their underlying data representation while leaving the properties invariant. It is proposed that a set of axioms be given to define the behavior of a data structure. Various forms of axioms are discussed. It is shown how axioms can be used to define the semantics of data definition facilities and in synthesizing data representations (perhaps automatically).

Advantages of separating property definition and representation are: abstract definition, proofs of properties of abstract objects, increased modularatiy, and postponement of data representation decisions, but principally data abstraction aids in the management of program complexity. Numerous examples of the axiomatic approach are presented.

Stonebraker75
Stonebraker, M. Implementation of integrity constraints and views by query modification. In SIGMOD75.

This paper indicates the mechanism being implemented in one relational system to prevent integrity violations which can result from improper updates by a process. Each interaction with the database is immediately modified at the query language level to one guaranteed to have no integrity violations. Also, a similar modification technique is indicated to support the use of "views," i.e. relations which are not physically present in the database but are defined in terms of ones that are.

Stonebraker77
Stonebraker, M. and Rowe, L. A. Observations on data manipulation languages and their embedding in general purpose programming languages. In VLDB77.

Many database query languages, both stand-alone and coupled to a general purpose programming language, have been proposed. A number of issues that various designs have addressed in different ways are treated in this paper. These issues include the specification of performance options, side effects, implicitness, the handling of types and the time of binding. In all cases, the emphasis is on a comparative analysis, rather than on an exhaustive survey of proposals. Several general observations on language design for database access are also made. (author's abstract)

Sungren74
Sungren, B. A conceptual foundation for the infological approach to data bases. In Klimbie, J.W. and Koffeman K. L. (Eds.), North-Holland, 1974.

Sungren78
Sungren, B. Data base design in theory and practice: Towards an integrated methodology. In VLDB78.

Sussman72
Sussman, G. J. and McDermott, D. V. The CONNIVER reference manual. AI Memo no. 259, MIT AI Lab., 1972.

Szolovits77
Szolovits, P., Hawkinson, L., Martin, W. A. An overview of OWL, a language for knowledge representation. MIT/LCS/TM-86, Lab. for Computer Science, MIT, 1977.

Taggart77
Taggart, W. M. and Thorp, M. O. A survey of information requirements analysis techniques. *ACM Computing Surveys 9*, 4 (Dec. 1977).

Thatcher79
Thatcher, J. W., Wagner, E. G., and Wright, J. B. Data type specification: Parameterization and the power of specification techniques. IBM Report RC 7757, July 1979. also in *Proc. SIGACT 10th Annual Conf. on Theory of Computation*, May 1978.

This paper extends our earlier work on abstract data types by providing an algebraic treatment of parameterized data types (e. g., sets-of-( ), stacks-of-( ), etc.), as well as answering a number of questions on the power and limitations of algebraic specification techniques. In brief: we investigate the "hidden function" problem (the need to include operations in specifications which we want to be hidden from the user); we investigate the relative power of conditional specifications and equational specifications; we show that parameterized specifications must contain "side conditions" (e. g., that finite-sets-of-d requires an equality predicate on d), and we compare the power of the algebraic approach taken here with the more categorical approach of Lehman and Smyth. (authors' abstract)

Towster79
Towster, E. A convention for explicit declaration of environments and top-down refinement of data. *IEEE Trans. Soft. Eng. SE-5*, 4 (July 1979).

Tsichritzis77
Tsichritzis, D. Research directions in database management systems. *SIGMOD Record 9*, 3 (1977), pp. 26-41.

Tsichritzis78
Tsichritzis, D. and Klug, A. (Eds.) The ANSI/X3/SPARC DBMS Framework. Report of the Study Group on Database Management Systems. *Info. Systems 3*, 4 (1978).

This is the final report of the ANSI/X3/SPARC Study Group on Database Management Systems which was chartered to investigate the potential for DBMS standardization. The report makes two strong statements. First, that the primary concern for DBMS standards should be the various interfaces rather than particular data models or representation details. The report goes to some length to describe these interfaces. Second, the report describes a framework or architecture commonly called the three schema architecture. The schemas are the internal schema, to be used to define storage details, the conceptual schema, to be

used to define in logical terms the entire database logically, and the external schema, to be used by a particular class of users to view the database in a particular application oriented manner. The report also addresses the problem of administrating such a system. The report supports the current trend towards providing users with an abstract view of the database which is appropriate for their particular needs.

Utah76
Proc. Conference on Data: Abstraction, Definition and Structure in *SIGPLAN Notices*, Vol. II 1976 Special Issue and in *SIGMOD FDT 8*,2 (1976).

Ullman80
Ullman, J. D. *Principles of Database Systems.* Computer Science Press, Potomac, MD., 1980.

Vassiliou79
Vassiliou, Y. Null values in database management: a denotational semantics approach. In SIGMOD79.

Verhofstad78
Verhofstad, J. S. M. Recovery techniques for database systems. *ACM Computing Surveys 10*, 2 (June 1978).

Warnier79
Warnier, J. D. *Logical Construction of Programs.* Van Nostrand Reinhold, New York, 1974.

VLDB75
Proc. 1st Int'l. Conference on Very Large Data Bases, Framingham, Mass., Sept. 1975. Available from ACM, New York.

VLDB77
Proc. 3rd Int'l. Conference on Very Large Data Bases, Tokyo, Japan, Oct. 1977. Available from ACM, New York.

VLDB78
Yao, B(Ed.), *Proc. 4th Int'l. Conference on Very Large Data Bases*, West Berlin, West Germany, Sept. 1978. Available from ACM, New York.

VLDB79
Proc. 5th Int'l. Conference on Very Large Data Bases, Rio de Janiero, Brasil, Oct. 1979. Available from ACM, New York.

Wasserman78a
Wasserman, A. I. A software engineering view of database management. In VLDB78.

This paper examines the field of database management from the perspective of software engineering. Key topics in software engineering are related to specific activities in database design and implementation. An attempt is made to show the similarities between steps in the creation of systems

involving databases and other kinds of software systems. It is argued that there is a need to unify thinking about database systems with other kinds of software systems and tools in order to build high quality systems. The programming language PLAIN and its programming environment is introduced as a tool for integrating notions of programming languages, database management, and software engineering. (author's abstract)

Wasserman78b
Wasserman, A. I., Sherertz, D. D., and Handa, E. F. Report on the programming language PLAIN. University of California, SF, Lab. of Medical Information Science TR 34, 1978.

This report describes a new programming language called PLAIN, intended for the systematic construction of interactive programs. PLAIN draws heavily on PASCAL for its structure and for many of its basic features. As a result the format closely follows those of PASCAL and EUCLID. The language is currently being implemented. Some important features of the language are: database variables and types, encapsulated data types, parameterized types, explicit type conversion, built-in and programable exception handling, and database expressions and operations (e.g., select, project, join, set ops.)

Currently, PLAIN is one of the few high level languages designed to support data abstraction and high level programming language access to a relational database. It is argued that abstract data types provide many advantages for databases including: constrained operations, semantic integrity, and database design techniques.

Wasserman79a
Wasserman, A. I. Principles of systematic data design and implementation. *Proc. INFOTECH Conference: Beyond Structured Programming*, Atlanta, Ga., April 1979.

The author points out the imbalance between procedural decomposition and the proper use of control structures versus data decomposition and the selection of data structures in software engineering. He emphasizes the much neglected aspects of data design. A unified approach to program and data design is proposed as essential to improving the quality of software.

The author provides a number of principles in the context of the software life cycle.
1. Apply systematic analysis methods to data as well as to procedures.
2. Identify the operations on data objects as well as the objects themselves.
3. Create a data dictionary to include all known constraints.
4. Defer low level data design decisions.
5. Employ information hiding to minimize the scope of effect.
6. Develop a library of data structures and operations.
7. Use structured data and data abstraction concepts.
8. Restrict the use of global data.
9. Restrict the use of pointers and other dynamic structures.

The author surveys a number of techniques which can be used to support these principles. The main message is that database and programming language concepts must be combined to improve software quality and that data abstraction is potentially a focus of this unification.

Wasserman79b
Wasserman, A. I. and Stinson, S. K. A specification method for interactive information systems. In PSRS79.

An approach for developing specifications for interactive information systems is discussed. An interactive information system is defined to provide naive users with conversational access to stored data through a predetermined set of operations. The system structure may be viewed as a user interface, a database, and a set of operations on the database.

Three logical components of the system specification are identified: a user view, a design view, and a correctness view. Each component links the specification with a different view of the software life cycle.

The paper describes the different views of a specification and a method of specifying the system structure. A specification method is presented which combines an informal specification with a more formal approach using transition diagrams for the user interface, logical database design methods for the data, and a data manipulation language for describing the operations. (from authors' abstract)

Wasserman79c
Wasserman, A. I. The data management facilities of PLAIN. In SIGMOD79.

The programming language PLAIN has been designed to support the construction of interactive information systems within the framework of a systematic programming methodology. One of the key goals of PLAIN has been to achieve an effective integration of programming language and database management concepts, rather than either the functional interface to database operations or the low-level database navigation operations present in other schemes. PLAIN incorporates a relational database definition facility, along with low-level and high-level operations on relations. This paper describes those features informally, showing how database operations are combined with programming language notions such as type checking, block structure, expression evaluation, and iteration. A brief description of implementation status is included. (author's abstract)

The objectives of PLAIN, beyond providing an interactive programming environment, include support for data abstraction, system modularity, program readability, testing and/or verification of programs, and the imposition of greater discipline on the programmer. Features of PLAIN include: string handling, pattern specification for I/O checks, exception handling facilities, database management, and systematic programming practices.

The author describes how strong type checking in PLAIN can be used to ensure a degree of semantic integrity. Also, relation types can be used as a basis for constructing abstract data types so as to provide users with constrained high-level operations over abstract objects.

Wasserman79d
Wasserman, A. I. The extension of abstract data types to database management. In preparation.

Waterman79
Waterman, D. and Hayes-Roth, F. (Eds.) *Pattern-Directed Inference Systems*, Academic Press, 1979.

Weber76
Weber, H. The d-graph model of large shared data bases: a representation of integrity constraints, and views as abstract data types. RJ1875 IBM San Jose, Nov. 1976.

The paper presents a model which offers a uniform notation to describe basic data structures like domains and relations, integrity constraints and views. The basic entities in the model are objects. Objects are characterized by types. With the type specification, we define the composition of objects out of other

objects of different types and manipulation rules for objects.

The concept of abstract data types is employed in the model which provides the encapsulation of data objects by all the operations applicable to these objects. The concept has been applied to model and implement the synchronization of concurrent accesses to shared resources in operating systems and for the design of programming languages which support structured and modular programming. It is shown here to be suitable to model integrity constraints and views and the manipulation restrictions imposed by constraints and views. (author's abstract)

Weber78
Weber, H. A software engineering view of database systems. In VLDB78.

This tutorial describes the nature of data abstraction, seen in terms of modules and abstract data types, and discusses the application of this concept to the development of database systems. The author discusses the benefits of data abstraction (separating implementation from specification, protection of data, the locality principle, and extensibility), the use of modules to implement data abstractions, and the interconnections and interface specifications of modules. The main theme of the tutorial is the use of data abstractions in the design of database systems. An extended example of a database application, an airport schedule, is used to illustrate the ideas. A language is presented for defining modules.

The author compares the traditional object model of a database with a module oriented view. Modular descriptions of a database and view implementation and communication using modules are discussed. The development technique is based on the author's D-graph model of databases, hence the discussion is in terms of both schemas and databases.

Wegbreit75
Wegbreit, B. The treatment of data types in EL1. *Comm. ACM 17*, 5 (May 1975), pp. 251-264.

Wegbreit76
Wegbreit, B. and Spitzen, J. M. Proving properties of complex data structures. *J. ACM 23*, 2 (April 1976), pp. 389-396.

Wells76
Wells, M. B. and Cornwall, F. L. A data type encapsulation scheme utilizing base language operations. In Utah76.

A data type encapsulation scheme in which the "space" operations are expressed naturally in terms of the base language operators is described. The scheme results from a conceptual separation of operators and procedure calls in the base language and produces a language of considerable expressive power. The scheme has been implemented and several examples are given. (author's abstract)

Wiederhold77
Wiederhold, G. *Database Design*. McGraw-Hill, New York, 1977.

Wielinga78
Wielinga, B. J. AI Programming Methodology. *Proc. AISB Conf.*, Hamburg, West Germany, June 1978.

The author argues that AI has outgrown LISP and that a new programming methodology and a new framework are needed. He argues that it is now time to develop a formal understanding of representation scheme building on the heuristic, experimental knowledge of the past. Structured programming is seen as difficult to apply to rather unstructured AI problems but that many benefits would be gained. Four aspects of programming languages are discussed: the abstract machine which determines the language semantics; the linguistic aspect which is an abstraction of the underlying machine; the programming environment for constructing and analyzing programs; and the understanding and description of programs. The main issue is seen as overcoming the complexity of AI problems and the main objective is to achieve "good" knowledge representation schemes.

A language framework is called for which includes: iteration control, control abstractions, data abstraction, database concepts, a programming environment, multiple views of data, formal definitions of representation schemes, a wide range of levels of understanding, and, of course, current AI concepts.

Winograd74
Winograd, T. Five lectures on artificial intelligence. Stanford AI Lab., AIM-246, 1975.

Winograd75
Winograd, T. Frame representations and the declarative/procedural controversy. In Bobrow, D. G. and Collins, A. (Eds.) *Representation and Understanding in Cognitive Science*, Academic Press, 1975, pp. 185-210.

The authors characterize the declarative/procedural controversy and propose that semantic networks attempt to achieve a synthesis of declarative and procedural knowledge.

A declarative representation of knowledge consists of a set of facts (which state *what*) plus general procedures to operate on them, e.g., axioms and proof procedures. This approach offers simplicity, fact independence, flexibility (a fact can be applied frequently but procedures are more specific), accessibility, and communicability (naturalness). A procedural representation is a collection of procedures which embody knowledge to state *how*. Procedural representations are more appropriate (simple, implementable) for some things; however, they are not as flexible as facts. They support second order knowledge (how to use facts), heuristic knowledge, and rich, powerful interactions.

Looking for a formalism to combine both, some knowledge declarative some procedural, may miss the fundamental ground for the dispute. There are simply different attitudes and approaches to the duality between modularity and interaction. Each problem must be decomposed in order to reduce the complexity of interactions. Both modular programming and fact compiling systems are attempts to achieve a synthesis through a concern for decomposition. These techniques attempt to take advantage of (declarative) decomposition without sacrificing possibilities for interaction.

Semantic networks and frames are used as a means of synthesizing declarative and procedural knowledge. In semantic nets generalization hierarchies are used to give full descriptions and to classify facts. Frames are used to focus attention on specific facts and to relate certain groups of facts explicitly. Important elements (frames) look like data types and indeed are related to abstract data types. To be useful such a scheme must have procedures attached to the network to operate on these elements. These procedures may also be structred in a generalization hierarchy from very general to very specific.

The authors conclude by noting that there is no formal difference between declarative and procedural approaches; however, each offers specific advantages. An important problem for frames is to include explicit control structures without loss of modularity.

Winograd79
Winograd, T. Beyond programming languages. *Comm. ACM 22*, 7 (July 1979), pp. 391-401.

The author contends that due to the increasing complexity of computer systems *higher* level programming languages than are currently envisaged are required. The main objective of these languages is not the origination of new independent programs but the integration, modification, and expansion of existing ones. The building blocks of future systems are to be modules and subsystems rather than individual instructions or sets of instructions. The major task to achieve this end is to provide a framework (knowledge base) and tools to deal with sets of interrelated descriptions of what results are desired.

Goals for such languages include: declarative rather than imperative descriptions; explicit definition of all constraints; multiple views of an application; procedureless programming (e.g., high level data structures that implicity include some control structures); uniform module descriptions stored in a centralized library of common system components; and specification of behavior.

To achieve these goals there is a need for: a descriptive calculus which is readily understandable and supports abstraction; a basis for describing processes (the system providing a knowledge base through its library of itself); the integration of structural and behavioral view points; and the integration of programming language systems, languages based on abstraction, and representation languages.

Although programming language, artificial intelligence, and database researchers all share the stated problems and goals, the author tends to ignore the contributions and objectives of the database community. Indeed, the subsystems in the database context could be database interfaces, external schemas, applications, or extensions of applications. The library concept is a synthesis of a program library and a database schema.

Wirth71a
Wirth, N. The programming language PASCAL. *Acta Informatica 1*, pp. 35-63, 1971.

Wirth71b
Wirth, N. Program development by stepwise refinement. *Comm. ACM 14*, 4 (April 1971), pp. 221-227.

This paper is mainly concerned with the problem of teaching people how to formulate programs. The author advocates a better method, which is more systematic and helps to give students an insight into the decisions that go into the making of a program.

The method is as follows: A nontrivial problem is taken, and the first step is to produce a solution in general terms, begging questions of detail. This solution is gradually refined and improved by filling in details at successively lower levels, each step in the process being the result of a conscious design decision, perhaps involving judgement of merits of alternative solutions. The process continues until a stage is reached such that the specification of the solution to the problem is at a low enough level to be encoded in a high-level programming language. Decisions about data formats would be made toward the end of this process, and would be taken in parallel with decisions about the more exact details of the problem solving method. At this stage, questions of efficiency in time and storage would arise. These ideas are illustrated by applying the method to the 8 Queens Problem.

Finally, it is concluded that if the method is applied skillfully, it will make the program that is its end product both portable and easy to maintain. The method is therefore applicable as a working tool as well as a tutorial tool. (CR21,631 J. P. Brown)

Wirth74
Wirth, N. On the composition of well-structured programs. *Computing Surveys 6*, 4 (December 1974) pp. 247-259.

This paper contains three sections and a conclusion. Section 1, entitled "Intellectual Manageability of Programs," describes the method of stepwise refinement in program development. Since programming tasks are growing more complex, it is essential to develop effective program management tools. Under the refinement technique, a program is regarded on an abstract level performing its tasks on abstract data and expressed in some suitable notation, possibly in natural language. The constituents of the program are then subjected to repeated refinement to lower levels of abstraction until a level is reached which is understandable to the computer, that is, a level written in a programming language. Clearly, the manageability criterion rests on the assumption that the constituent operations are sufficiently simple to connect and, in fact, should be developed as operations with single starting and ending points. Moreover, the composition of each constituent must also be simple.

The selection of the appropriate programming structures is the subject of Section 2, "Simplicity of Composition Schemes." Essentially, the idea is to use such statements as the WHILE and REPEAT and to shun GOTO's. The author stresses that

...well structured programs should not be obtained merely through the formalistic process of eliminating GOTO statements from programs that were conceived with the facility in mind, but that they must emerge from a proper design process...

To stress this point, Section 3, "Loop Structures," contains two examples which demonstrate that often the exit from the middle of the loop construct is based on preconceived notions rather than on a real need. Indeed, even better solutions can occasionally be obtained by adhering to the fundamental constructs described in Section 2. Finally, the author reiterates his previous arguments for well-structured programming language as widely available as, say, FORTRAN, in the conclusion.

This article is extremely well-written and certainly a welcome addition to the literature of good programming style. It should be required reading in every programming language course. (CR28, 101 S. Chen)

Wirth75
Wirth, N. An assessment of the programming language PASCAL. *IEEE Trans. on Soft. Eng. 1*, 2 (June 1975), pp. 192-198.

The evaluation of a given work by its own author is not a frequent matter in computer science, especially when this work has already given rise to some controversy. Thus, it is very interesting and refreshing to read critical comments on the programming language PASCAL, made by its own designer. This evaluation is made in the light of software reliability, and obviously does not concern only criticisms of the language; it is made in a very lucid and impartial fashion, which yields general guidelines for the design of programming languages and programming tools. The paper is easy and pleasant to read, and contains some well-coined sentences (often provocative), worthy of quotation at the end of this review.

The author begins by discussing the notion of program reliability, and suggesting that program correctness is a much more valid notion: "...the degree to which a program is unreliable is exactly the probability with which its incorrect parts are executed." A brief list and description is then made of the most important features of PASCAL which contribute to reliable programming by providing useful redundancy; these include the symbolic scalar types, record types, set types, subrange types, and the simple forms of iterative and selective statements. The rest of the paper discusses the main deficiencies of the language; some of these have already given rise to controversy, others have remained unnoticed by commentators without experience in the language. These latter points include operator precedence, the goto

statement, the restrictions on array bounds in declaration and parameter specification, the concept of file, and the notion of discriminated or free type unions.

The discussion is easy to follow, even by a reader unfamiliar with PASCAL (and equally interesting for that sort of reader), and the different choices within the language are well delineated. The author's conclusion is the main source for the following quotations:

It is probably the most disheartening experience for a language designer to discover how features provided with honest intentions are "ingeniously" misused to betray the languages's very principles....

Instead of relying too much on either antiquated "debugging tools" or on futuristic automatic program verifiers, we should give more emphasis to the systematic construction of programs with languages that facilitate transparent formulation and automatic consistency checks....

The urge to gain flexibility and "power" through generalizations must be restrained. A generalization may easily be carried too far and have revolutionary consequences on implementation....

In many cases, security and flexibility are antagonistic properties....

A rich language may be welcome to the professional program writer whose principle delight is his familiarization with all its intricate facets. But the interests of the program reader dictate a reasonably frugal language. People who want to understand a program (including their own), compilers, and automatic verification aids all belong to the class of readers. In the interest of increased quality to software products, we may be well advised to get rid of many facilities of modern, baroque programming languages that are widely advertised in the name of "user-orientation," "scientific sophistication," and "progress." (CR29,417 O. L. Lecarme)

Wirth77a
Wirth, N. MODULA: A language for modular multiprogramming. *Software - Practice and Experience 7*, 1(Jan. 1977), pp. 3-35.

This paper defines a language called Modula, which is intended primarily for programming dedicated computer systems, including process control systems on smaller machines. The language is largely based on Pascal, but in addition to conventional block structure, it introduces a so-called module structure. A module is a set of procedures, data types and variables, where the programmer has precise control over the names that are imported from and exported to the environment. Modula includes general multiprocessing facilities, namely processes, interface modules and signals. It also allows the specification of facilities that represent a computer's specific peripheral devices. Those given in this paper pertain to the PDP-11. (author's summary)

Modula itself has no direct relationship to database concepts, e.g., even Pascal's variant records and files are missing and there is no input/output facility. However, it is argued that these facilities can be programed and encapsulated within modules. Modula provides facilities for multiprogramming using the module concept. Modula also provides a so-calld interface module which corresponds to Hoare's monitors used to control critical sections.

Wirth77b
Wirth, N. The use of Modula. *Software, Practice, and Experience 7*, 1 (1977), pp 37-65.

Three sample programs are developed and explained with the purpose of demonstrating the use of the programming language Modula. The examples concentrate on the uses of modules, concurrent processes and synchronizing signals. In particlar, they all focus on the problems of operating peripheral

devices. The concurrency of their driver processes has to occur in real time. The devices include a typewriter, a card reader, a line printer, a disk, a terminal with tape cassettes and a graphical display unit. The three programs are listed in full. (author's summary)

Wirth77c
Wirth, N. Design and implementation of Modula. *Software, Practice and Experience 7*, 1 (1977), pp. 67-84.

Wong77
Wong, H. K. T. and Mylopoulos, J. Two views of data semantics: A survey of data models in artificial intelligence and database management. *INFOR 15*, 3 (Oct. 1977).

The goal of this paper is to establish that there exists a strong relationship between current issues of data models in database management and representations of knowledge in AI. The differennt data modelling techniques that have been used in the two research areas are surveyed and classified into predicate calculus-based, network, and procedural ones. The similarities and differences between them are presented. (author's abstract)

There is an emphasis on the case of predicate calculus and semantic networks to describe database semantics. The authors conclude that the various differences in goals and methodologies between the two areas will diminish the predict that current work is tending to merge the features of predicate calculus, networks, and procedural representations. E.F. Codd has said that this paper incorrectly equates the relational model with first order predicate calculus.

Wong79
Wong, E. and Katz, R. Logical design and schema conversion for relational and DBTG databases. In Chen79.

Woods75
Woods, W. A. What's in a link: Foundations for semantic networks. *Representation and understanding*, Bobrow and Collins (Eds.), Academic Press, 1975.

Wortman79
Wortman, D.B. On legality assertions in Euclid. *IEEE Trans. Soft. Eng. SE-5*, 4(July 1979).

Wulf76a
Wulf, W. and Shaw, M. Global variables considered harmful. *SIGPLAN Notices 8*, 2 (Feb. 1976).

Wulf76b
Wulf, W. A., London, R. L., and Shaw, M. An introduction to the construction and verification of ALPHARD programs. *IEEE Trans. on Soft. Eng. SE-2*, 4 (Dec. 1976).

Yeh77a
Yeh, R. T. (Ed.) *Current Trends in Programming Methodology*. Prentice Hall, Englewood Cliffs, N.J.
Vol. I: *Software Specifications and Design*, 1977.
Vol. II: *Program Validation*, 1977.

Yeh77b
Yeh, R. T. and Baker, J. W. Towards a design methodology for DBMS - a software engineering approach. In VLDB77.

A design methodology for DBMS is presented. The design process is captured in three interacting models: one for system architecture; one for hierarchical performance evaluation; and one for design structure documentation. These three models are developed concurrently through a top down design process. The authors claim that through their methodology the design is evaluated and its consistency is checked during each stage of the design process. Advantages of this methodology are that the resulting systems are: reasonably independent of their environments, reliable, and easily modifiable. A small example is used to illustrate the methodology. (R. T. Yeh)

Yeh78a
Yeh, R. T., Roussopoulos, N., and Chang, P. Database design - An approach and some issues. SDBEG-4, U. Texas, Austin, 1978.

This paper describes some inadequacies of current database design methodologies and presents a new approach. The approach includes mechanisms for: expressing and formalizing user requirements and system specifications, consistency and completeness checking, performance evaluation, and the modification of database structures. The approach is based on artificial intelligence and software engineering concepts such as abstraction, successive refinement, modularity, semantic networks, and frames. The main emphasis is the design of conceptual schemas.

Yeh78b
Yeh, R. T., Araga, A., and Chang, P. Software and database engineering - towards a common design methodlogy. SDBEG-6, Computer Science, University of Texas, March 1978.

The authors argue that the areas of software engineering and database management have much to benefit from each other, in particular, a unified design methodology. However, there has been little cross-fertilization. Current methods can aid to resolve some issues while a clear research direction can be seen for a unified approach.

Three major weaknesses of current software methodologies are described: a lack of a separation of concerns, inadequate design evaluation, and overemphasis on the "functional" approach. Two problems in database management are a lack of well-structured DBMS's and a need for a design methodology aggrevated by the inherent complexity of database applications (leading to incompleteness and inconsistency) and the inability to enummerate and evaluate alternative designs.

In particular, the authors claim that a lack of a good method led to: design errors, high maintenance costs, impossibility of top down design; extreme difficulty of validation, and no specifications for user guidance.

The authors propose that these problems can be addressed through an appropriate unification of concepts in the two areas based on the notion of a "conceptual model." A verified design methodology is proposed and discussed. The methodology is applied to database systems in terms of a multi-level design.

The most important concept here is that the design and implementation of database applications are large software development projects for which many techniques exist.

Yourdon79
Yourdon, E. and Constantine, L. L. *Structured Design: A discipline of computer program and systems design*. Prentice-Hall, 1979.

Zelkowitz78
Zelkowitz, M. V. Perspectives on software engineering. *ACM Computing Surveys 10*, 2 (June 1978).

Software engineering refers to the process of creating software systems. It applies loosely to techniques which reduce high software cost and complexity while increasing reliablity and modifiability. This paper outlines the procedures used in the development of computer software, emphasizing large-scale software development, and pinpointing areas where problems exist and solutions have been proposed. Solutions from both the management and the programmer points of view are then given for many of these problem areas. (author's abstract)

Zelkowitz79
Zelkowitz, M. V., Shaw, A. C. and Gannon, J. D. *Principles of Software Engineering and Design*. Prentice-Hall, Englewood Cliffs, N.J., 1979

Zilles75
Zilles, S. N. Abstract specifications for data types. IBM San Jose, 1975.

NBS-114A (REV. 9-78)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET	1. PUBLICATION OR REPORT NO. SP 500-59	2. Gov't. Accession No.	3. Recipient's Accession No.

4. TITLE AND SUBTITLE

COMPUTER SCIENCE AND TECHNOLOGY:
Data Abstraction, Databases, and Conceptual Modelling:
An Annotated Bibliography

5. Publication Date May 1980

6. Performing Organization Code

7. AUTHOR(S)

Michael L. Brodie

8. Performing Organ. Report No.

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Department of Computer Science
University of Maryland
College Park, MD 20740

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

National Bureau of Standards
Department of Commerce
Washington, DC 20234

13. Type of Report & Period Covered Final

14. Sponsoring Agency Code

15. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 80-600052

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

16. 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 bibliography contains entries for over 350 books, articles, and papers on issues within the area of conceptual modelling of dynamic systems of complex data. The entries have been drawn from recent work in the areas of database management, programming languages, artificial intelligence, and software engineering. The bibliography has two purposes: to present a comprehensive list of annotated references to research into issues of data abstraction, databases, and conceptual modelling: and second, to encourage the cross-fertilization of the three research areas of database management, programming languages, and artificial intelligence.

17. KEY WORDS *(six to twelve entries; alphabetical order; capitalize only the first letter of the first key word unless a proper name; separated by semicolons)*

Abstract data types; Artificial intelligence; Data abstraction; Database management systems; Data structures; Programming Languages; Software engineering

18. AVAILABILITY ☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☒ Order From Sup. of Doc., U.S. Government Printing Office, Washington, D.C. 20402

☐ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

19. SECURITY CLASS (THIS REPORT)	21. NO. OF PRINTED PAGES
UNCLASSIFIED	86
20. SECURITY CLASS (THIS PAGE)	**22. Price**
UNCLASSIFIED	$3.75

USCOMM-DC

Order From Sup. of Doc., U.S. Government Printing Office, Washington, D.C. 20402

$3.75

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SCIENCE & TECHNOLOGY

Superintendent of Documents,
Government Printing Office,
Washington, D. C. 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Bureau of Standards Special Publication 500-.

Name _____

Company _____

Address _____

City _____ State _____ Zip Code _____

(Notification key N-503)

☆ U.S. GOVERNMENT PRINTING OFICE : 1980 O-311-065 (82)

NBS TECHNICAL PUBLICATIONS

PERIODICALS

JOURNAL OF RESEARCH—The Journal of Research of the National Bureau of Standards reports NBS research and development in those disciplines of the physical and engineering sciences in which the Bureau is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Bureau's technical and scientific programs. As a special service to subscribers each issue contains complete citations to all recent Bureau publications in both NBS and non-NBS media. Issued six times a year. Annual subscription: domestic $17; foreign $21.25. Single copy, $3 domestic; $3.75 foreign.

NOTE: The Journal was formerly published in two sections: Section A "Physics and Chemistry" and Section B "Mathematical Sciences."

DIMENSIONS/NBS—This monthly magazine is published to inform scientists, engineers, business and industry leaders, teachers, students, and consumers of the latest advances in science and technology, with primary emphasis on work at NBS. The magazine highlights and reviews such issues as energy research, fire protection, building technology, metric conversion, pollution abatement, health and safety, and consumer product performance. In addition, it reports the results of Bureau programs in measurement standards and techniques, properties of matter and materials, engineering standards and services, instrumentation, and automatic data processing. Annual subscription: domestic $11; foreign $13.75.

NONPERIODICALS

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NBS under the authority of the National Standard Data Act (Public Law 90-396).

NOTE: The principal publication outlet for the foregoing data is the Journal of Physical and Chemical Reference Data (JPCRD) published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements available from ACS, 1155 Sixteenth St., NW, Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

Order the **above** *NBS publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*

Order the **following** *NBS publications—FIPS and NBSIR's—from the National Technical Information Services, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Services, Springfield, VA 22161, in paper copy or microfiche form.

BIBLIOGRAPHIC SUBSCRIPTION SERVICES

The following current-awareness and literature-survey bibliographies are issued periodically by the Bureau:

Cryogenic Data Center Current Awareness Service. A literature survey issued biweekly. Annual subscription: domestic $25; foreign $30.

Liquefied Natural Gas. A literature survey issued quarterly. Annual subscription: $20.

Superconducting Devices and Materials. A literature survey issued quarterly. Annual subscription: $30. Please send subscription orders and remittances for the preceding bibliographic services to the National Bureau of Standards, Cryogenic Data Center (736) Boulder, CO 80303.

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Washington, D.C. 20234

OFFICIAL BUSINESS

Penalty for Private Use, $300

POSTAGE AND FEES PAID
U.S. DEPARTMENT OF COMMERCE
COM-215

SPECIAL FOURTH-CLASS RATE
BOOK