PANDA: An Object-Oriented Database Based On User-Defined Abstract Data Types

Max J. Egenhofer
Andrew U. Frank

Report No. 67

ABSTRACT

Existing database systems do not fulfill the requirements and needs of engineering applications such as CAD/CAM or cartography. We report on solutions which were integrated into the PANDA database management system, suited for such a use. PANDA supports an object-oriented program design and is based on the concept of a database kernel. It has been successfully used in research and teaching for several years.

The paper stresses the consequences these requirements had on the modularization of the software. It presents a software-environment suitable for programming in the large and shows how software engineering concepts influenced the database design.

An object-oriented data model is introduced which is based on user-defined abstract data types and applies methods for data structuring. All operations deal with objects and no lower structured items (such as attributes) are visible. In order to be stored in the database, each abstract data type must contain certain operations required to access objects. Most of these access-operations are very similar and type-independent. We will show that it is advantageous to internally create object-types using combintaions of lower-structured parts: this helps to avoid redundant coding and allows the reuse of object parts.

1 INTRODUCTION

Existing 'commercial' database systems are not sufficient for applications to engineering tasks such as CAD/CAM, or for cartographic and geographic information systems. Systems for CAD/CAM or for space-related information (Geographic or Land Information Systems [Frank 1983]) integrate a database system into a much larger program system. A deficiency we have observed with the standard database management system is the lack of support for modular program design. Following the concept of abstract data types [Guttag 1977] [Parnas 1978] [Zilles 1984], each module should encapsulate an object type with all its pertinent operations. Standard database systems, based on networks [CODASYL 1971] or the relational [Codd 1982] data model did not fit easily into this concept. A number of methods of capturing more semantics in the data model have been proposed in the literature (for an overview and critique see [Brodie 1984] which contains a large number of references), but most of these methods have not been implemented, and none of them is easily available.

Complex tasks require complex structuring tools. 'Complex', however, need not mean 'complicated': the support of data structuring must be powerful without sacrificing ease of use, extensive without becoming excessive. Our approach was to decide on a single method of solving a typical problem and to provide a tool to solve it this way - rather than to provide for several alternatives, which too often only confuses the programmer.

This paper does not specifically discuss some of the detail problems, but rather concentrates on the overall design and the lessons we learned from applying software engineering principles to database design. Moreoever, it does not rely on some hypothetical design, but rather is based on experiences acquired during the design and implementation of PANDA. The PANDA database system has been running for several years and is primarily used for research and teaching. It has been revised a number of times, each time it became simpler and clearer.

This paper begins with a brief overview of the requirements we felt existing database systems lacked. It then presents the software engineering methods we decided to use for our LIS/GIS-implementation.

Subsequently, a data model is introduced which is suitable for application to object-oriented design. We will discuss the functionality of a storage- and retrieval-system of a database. This sub-system is completely general and does not depend on the type of data stored. As it does not deal with the semantics of data, it seems too limited in functionality to be called 'database kernel'.

The object-model was extended by the addition of access paths. The internal operations will be discussed. In order to reuse abstract data types, especially for the access-operations, objects are presented as components of abstract data types. The paper closes with a minimal set of operations an abstract data type requires to be recognized as part of an object.

2 DATABASES FOR SPATIAL INFORMATION SYSTEMS

Members of the CAD/CAM-community complain that there are no database systems which are appropriate for their demands [Buchmann 1985], [Sidle 1980], [Udagawa 1984].

They state that standard database systems do not fulfill the requirements of engineering applications in terms of performance [Stonebraker 1982], [Wilkins 1984], [Haerder 1986], [Maier 1986], treament of complex objects [Lorie 1983], [Brodie 1984b], [Batory 1984], appropriate data structuring [Johnson 1983], and have suggested that there is probably no universally applicable solution [Batory 1986].

Only recently, design of databases for spatial information system has become a research topic and several proposals [Lipeck 1986] [Scheck 1986] have been published. [Aronson 1983] has discussed software engineering methods used in the ARC/INFO design, but ARC/INFO, like other older GIS software, does not use database principles for the management of geometric data.

After several years of experimenting with a commercially available database management system, which followed the CODASYL [CODASYL 1971] document, we built our own system. We felt that the effort of working with tools geared toward a different class of uses was so difficult and time-consuming that a more appropriate system would allow faster progress.

Our list of goals included such principles as:
- the management of very large amounts of data without performance penalty and

- support of spatial storage clusters and spatial access to provide fast response time [Frank 1983a]. This would permit us to manage large spatial data collections without subdividing them into several user-visible map sheets ('a single database for the whole of the users world')
- no physical separation of the geometric from the non-spatial data in different databases, but rather an integration of both types of data into one system. This would require
- user-defined complex structured types in the database
- a buffering schema to reduce physical disk access; exploiting the geometric neighborhood and the specific behaviour of a system with primarily graphical output

3 A DATABASE WITHIN AN ENVIRONMENT FOR PROGRAMMING IN THE LARGE

A database is only one part in the large effort to produce a CAD/CAM system, a land information system, or any of the other non-standard applications. It is thus necessary that database methods and techniques fit within a software engineering paradigm. We have selected a theoretically founded method for modularization based on abstract data types. We then constructed a support system which takes care of the mechanics but also controls and enforces the method.

3.1 Programming In The Small Using Pascal

In 1982, when this effort started, Pascal was the only modern language which was available on most computers. We decided to use Pascal because
- it was reasonably standardized
- transporting code to other machines was possible
- high quality compilers and run time systems were available
- regular courses for learning the language were offered
- we were already familiar with this language

Using Pascal as the main programming language also allowed use of the database management system in undergraduate teaching.

Portablility was achieved by conditional compilation features which allow insertion of machine-specific code within machine-independent code. Our experience with this has been good: without major changes code was transferred from such vastly different hardware and operating systems as DEC-10 (under TOPS-10) to IBM 370 (under VM/CMS) and to VAX and microVAX (under VMS and MicroVMS, respectively). With conditional compilation and the isolation of machine-dependent features in special modules, a transfer did not require undue effort.

3.2 Tools For Programming In The Large

Large programming efforts must be broken into separate modules. A major goal is to design modules which are as independent as possible. They should be capable of being built by different individuals with minimal interaction. This is extremely useful in a research environment, where code has to be changed quite frequently, or for teaching, where students can combine existing modules for various tasks without starting from scratch each time.

Along with the benefits of Pascal, however, we inherited all its restrictions. The most limiting issue was that standard Pascal does not include independent compilation. The syntax used to achieve independent compilation is thus very different in different Pascal compilers. We therefore had to build a precompiler which satisfied varying requirements.

A major stumbling block for independent compilation using Pascal is the common demand that the procedure or function heads of all used external procedures are included in the compilation unit, in order for the compiler to check the types of the parameters passed. Our precompiler automatically copies the necessary procedure heads from the defining compilation unit to the using unit, thus assuring type conformity across modules. This is very similar to the methods used for Ada or Modula-2 compilers [Wirth 1982], [Ada 1983].

3.2.1 Abstract Data Types

The software engineering method is based on the work on formal specifications using abstract data types. An abstract data type (ADT) or a multi-sorted algebra is a mathematical structure which fully defines the behaviour of objects, i.e., the semantics of the object-type and its operations. Abstract data types are specified as a theory describing what sorts of objects (types) are dealt with and what kinds of operations they are subject to. A set of axioms determines the effects of the operations.

It is up to the designer to assure that the sorts and their operations form a reasonable and meaningful object.

Abstract data types are very useful in the design of large systems. They allow a much more comprehensive and complete way of specifying routines together with their axioms than an extensive programming language does.

Abstract data types can be combined in layers, where higher-level abstract data types are first described independently (with a set of axioms to represent a theory) and it is then shown how this behaviour can be achieved using other, hierarchically lower abstract data types (called an 'abstract implementation' [Olthoff 1985], [Frank 1986]). Such abstract implementations can be formally checked for correctness, by proving that the axiomatically specified behaviour of the upper abstract data type follows from the abstract implementation and the axiomatically specified behaviour of the lower abstract data type.

3.2.2 Support System

In software engineering, abstract data types may be represented by modules which encapsulate a type and its operations.

Facilities such as packages and use clauses in Ada or modules and import commands in Modula-2 or in our precompiler allow the programmer to easily translate abstract data types into executable code.

Our precompiler for Pascal unfortunately does not allow coding of generic or parametrized abstract data types [Cardelli 1985] in single modules. Thus, several modules have to be written with only slight differences in the code.

We have found that effective use of a software engineering methodology is only possible with a comfortable, automated support system. Programmers will conform to a standard if it is made easy; our Pascal precompiler allows essentially only one method of modularization and enforces coding standards, but advantages for programmers are so obvious that these restrictions are not resented.

4 AN OBJECT-ORIENTED DATA MODEL

Our data model is built on the basic concepts of abstraction: classification, generalization, and aggregation. We further add the notion of access path to the data model, as we observe that access methods are among the most important functions in a database kernel [Haerder 1986].

4.1 Objects

We will use the word 'object' to describe a single occurrence (instantiation) of data describing something that has some individuality, some observable behaviour. Objects have some operations associated with them, which can be carried out upon them. The terms 'sort', 'type', 'abstract data type', or 'module' will be used to refer to types of objects, depending on the context.

4.2 Abstraction Methods

The abstraction methods are similar to the ones presented in [Brodie 1984]. We recognize three methods which can be combined to structure complex objects and their components.

4.2.1 Classification

Classification can be expressed as the mapping of several objects (instances) to a common class, similar to types and instances in programming languages. All objects of a common class share the same operations. Classes translate to abstract data types or modules in the implementation. Every object is an instance of a class (is_a-relation).

4.2.2 Generalization

Several classes of objects which have some operations in common are grouped together into a more general 'superclass' [Goldberg 1983]. Objects of the lower class inherit all the operations (or attributes) which are in common from the superclass. We recognize only hierachical inheritance betweeen classes and superclasses.

4.2.3 Aggregation

Several objects can be combined to form a semantically higher-level object where each sub-object (part) has its own functionality.

Aggregation establishes a part-of relation between objects.

4.3 Implementation

Classes are coded as Pascal records with corresponding operations encapsulated in a module. Objects are variables of this type. Superclasses are also built as modules with record types consisting of variant parts, one for each different subclass. All the common operations in the superclass are exported by this module.

Entities are Pascal records with variant parts. They represent the different classes.

Several entities can be aggregated to form a more complex type by connecting them with CODASYL-type sets, here called aggregates.

To access inner parts of an aggregate object is an operation on this object, which stands not only for itself (the CODASYL view) but also represents all the objects aggregated with it. Operations on the parts aggregated with an object are then executed in a 'for each' loop [Liskov 1981] (implemented with a 'while' construction). This seems to be the solution which fits best within the framework of a statement-oriented language like Pascal, whereas functional language could offer other concepts [Backus 1978].

4.4 Assessment

This concept is similar to the requirements established in [Dittrich 1986]: it is structurally, operationally, and behaviourally object-oriented. The limitations imposed by the use of a statement-oriented, traditional programming language like Pascal could be mostly overcome by the use of the precompiler for separata modular compilation. This concept, however, does not include a message-passing paradigm but relies on function calls. Message-passing is often cited as necessary for object-oriented design, but in our opinion it is not an essential feature.

5 FUNCTIONALITY OF A DB STORAGE AND RETRIEVAL SUBSYSTEM

We applied our software methodology not only to the desing of the database interface but also to the internal design of the database management systems [Dittrich 1986]. Using modular concepts there moves us toward the goal of database management systems which can be combined from different components depending on the applications requirement. In a first group we consider basic building blocks for storage and retrieval of data.

5.1 Disk Storage Sub-system

A database must provide persistent storage for objects modelled in the information system. Most of the applications considered will create very large data collections which will require use of magnetic or optical disk systems for permanent storage. Most operating systems do not allow the user to directly access disk storage, but rather allow access through the file system. Operations required to store and retrieve data using such a file system are

- create a file with a given name
- make an existing file accessible and cancel access (open and close)
- read a specific part of a file
- write something to the file

Database management systems use random access for reading and writing an arbitrary block. Disk access is expensive, thus input and output are buffered with a least-recently-used-strategy.

5.2 Transaction Support

In order to keep complex data structures consistent, a transaction concept is necessessary, providing for the extensions of the classical properties atomicity, consistency, isolation, and durability (ACID) [Haerder 1985].

5.3 Support For Multiple Users And Distributed Systems

Central databases are important resources and access to them must be available for many operations in parallel and at the same time.

For organizational reasons, large databases are often not centralized, but distributed over several computer systems. This may improve their availability during local hardware failures, reduce data transfer cost, etc. In our opinion, support for distribution should be built into this layer of the subsystem.

5.4 Entity-storing Sub-system

The database does not deal with blocks or pages. It rather works on the objects we define using abstract data types with corresponding operations. All abstract data types defined by the user have in common
- to be of a generalized object type ('storage_element'),
- and then the operations 'store' and 'retrieve'.

All objects are treated as if their type were a subtype of a general type 'entity'. This general type exports the basic operation of the storage sub-system. The access operations of this high-level abstract data type are defined as follows:

```
abstract data type: 'storage_element'
operations:

STORE    : storage_element x length  ->  felp
RETRIEVE : felp -> storage_element
```

with the axiom

```
RETRIEVE (STORE (s, length)) = s
```

Felp stands for 'file element pointer' and is a unique tuple identifier for each storage element. It is probably useful to add operations to MODIFY and DELETE a storage-element, but it does not alter the basics.

The storage sub-system does not know anything about other operations owned by the objects or their internal structure. It only deals with storing and retrieving of entities.

Improvements in performance can be achieved by reducing the number of disk accesses by bufferinng the storage elements.

5.5 Assessment

This subsystem is very important for a database management system. It is obvious that it must be built in a modular fashion, such that components may be changed in order to deal with other specific requirements. However, this part does not deal with the semantics of data in any way, and thus seems not to warrant the label 'database kernel'.

6 THE DATA MODEL MAPPED INTO THE DATABASE

The database must include the ideas represented by the data model. This part of the system has the task of combining the user's view of the database (including different objects and special operations) with the generic database system.

The major difference of this model from previously proposed so-called semantical or functional data models [Shipman 1981], is the absence of the notion of attributes in entites. In our view, attributes are an internal part of an entity definition and thus hidden, not accessible to the database. However, the defining module for the entity must export some operations to be used by the database if support for a special access method is desired.

6.1 Managing Meta Data

With the data definition language (DDL) the user defines the meanings of the objects and their combinations which the database is to handle. The manager for the meta-data has the task of simply translating the semantics expressed in the data defintion language into a form which can be systematically processed. He filters the meaningful information and only leaves a schematic form visible to the database. Some simple checks can be done to detect obvious errors in the data definition language.

6.1.1 Entities

Entities are the basic abstraction upon which our object-oriented data model is founded. All the other steps of modelling are related to or based on entities. Entities are all of the same super-type, a generalized object type the database deals with.

Entity types are defined by the user as modules (Pascal record types with operations). Their internal structure is of no relevance on this level of the data model.

6.1.2 Aggregates

Aggregates are formed with links between entities (1:m-relations). They are used to establish aggregations, meaningful connections between objects.

6.1.3 Access Paths

Paths describe how to access objects with either specified values or within ranges of values. Access paths are defined for object types which need to export certain operations to be used by the access mechanisms.

Accesses are rather necessary, as sequentially searching through the whole database is too expensive and unacceptable for performance reasons. A database kernel must, therefore, offer a number of methods for higher performance access. Each access mechanism requires internal specific support for both storage and access by some searching algorithm [Knuth 1973]. These structures have to be maintained and managed by the database for each path separately, i.e., any change in the data set implies checks and modifications on the access structures. The structures must be consistent and their maintenance embedded in the transaction concept.

Most access mechanism needs to reorganize its structure after a number of modifications in the database. It would be advantageous if the system could manage the reorganization by itself without notably decreasing performance.

We have included several different types of access paths, but others may be added according to application needs. Each path requires its special, internal structure and performs a different access-function in a higher object-oriented interface.

6.1.3.1 Sequential Access

The sequential access is the trivial case of an access path. The internal structure, if any, is an aggregate collecting all the existing objects.

6.1.3.2 Access To Objects Using Unique Keys

We will call access by means of unique keys 'direct access' as it allows to retrieve single objects directly, unrelated to any other object. This access path allows retrieval of objects if a unique identifier is known, and adds a consistency constraint to the database.

In applications, this access-method is likely to be the most frequently used for starting work. It requires exact knowledge of one part of a single object.

The restriction of working with a unique key sometimes causes this access path to be problematic. The requirement of unique keys is a very strict constraint, which in reality is often an idealized assumption. It does not allow duplicates, i.e., no two or more objects in the database may have the same value. This consistency constraint has to be enforced by the sorting algorithm. As an experiment we added a generalized direct access path where non-unique key values are permitted.

In our present implementation, direct access is supported by a hashing algorithm and tables. Two operations are required to support this path: (1) calculating a hash-value and (2) checking for equality. Both operations are needed, as several different instances may have the same hash-value. Direct access is functionally composed of calculating a hash-value and then identifying the desired object among the objects owning the same hash-value. Of course, the two operations are correlated, since being identical implies having the same hash-value (but not the reverse).

6.1.3.3 Access Using Unique Keys Within Aggregates

We will call the access using unique keys within aggregates 'access within aggregates' as it allows retrieval of an object as part of a higher construct.

Access within an aggregate is semantically less strict than direct access, as keys must only be unique within the aggregate. However, its supporting structure has to enforce the uniqueness of identifiers this path is defined on, too.

Indirect access can be structured with B*-trees where the super-part is the root and the sub-parts are in the leaves, sorted on some specified criterion. This path uses two operations: a monotone mapping of the value to an integer which is the sorting criterion within the B*-tree and a check for equality.

6.1.3.4 Sorted Access On Aggregates

Sorted access allows retrieval of the sub-parts of aggregates in a sorted sequence. The sequence is fixed by a 'before'-operation which includes the sorting-criteria for the object. Generally, the sorted access can be based on the same structure as the access within aggregates.

6.1.3.5 Spatial Access

As opposed to the access paths above, spatial access is a range access. The choice of objects is not based on equality but on the criterion of lying within a range or overlapping with another spatial extension. This requires an operation which checks whether a given object lies between some upper and lower limits.

For the support of spatial objects, there are quite a large number of proposals for different structures [Tamminen 1981] [Nievergelt 1981] [Frank 1983a] [Guttman 1984]. The spatial access path is based on checking whether an object lies within a certain spatial extension. In our case [Frank 1983a], the spatial structure is based on a minimal bounding rectangle each spatial object owns. Therefore, the operation for spatial access checks whether the bounding rectangle and the area overlap.

Very similar to aggregates, searching within a range results in a set of objects requiring a 'for each'-loop to access the whole set of included objects.

7 ENRICHING THE DATABASE WITH VALUE-ABSTRACT DATA TYPES

A pathless database requires only one aspect of the abstract data types, namely the type-definition, i.e., the length of the entity-record. The database can perform all the operations such as store, aggregate, and delete just on the object types. No knowledge is required as to how to operate on object-details.

For the concept of paths, the database needs to have some specific operations for the abstract data types the paths work on. The different path-structures need to know what operation should be used to establish the structure. These operations, of course, are part of the abstract data types. Thus, the task is to combine a standardized managing mechanism (the database) with some user-defined abstract data types.

Very similar (or even the same) code must be written to provide the operations for the access paths which are dependent on the object-type. Each spatial object-type, for example, requires a routine test to see whether it lies within a specified region or not. All these routines have the same structure, in fact: they act on a part of the object with the same type, regardless whether the object is a node or an area. It is appropriate to consider types as abstract data types defining value types which can be combined to compose entity types.

7.1 Attributes

We have noticed that not every object-type has its own, independent characteristics. A great many of them are based on a very small set of abstract data types which are repeatedly used in different combinations. The principle of reusing software atoms [Batory 1986] is crucial to object-oriented databases. However, writing the same hashing algorithm, for example, for different object-types several times does not support reusability. A better way has to be found to compose objects out of parts which can be used in several object-types.

Most object types are composed of less complex parts which represent properties and values. All these value-ADTs together form a set of abstract data types objects can be composed of. This 'toolbox' can arbitrarily be extended with any new user-defined abstract data type by either combining existing abstract data types to more complex and powerful ones or creating a new abstract data type.

This leads to introducing attributes as parts of object-types. One attribute can contribute to several different object-types. The object-oriented concept is not violated by the attributes. Attributes must rather be seen as a tool to avoid highly redundant and similar coding. The operations still work exclusively on objects.

7.2 Values: The Abstract Data Types For Attributes

Abstract data types objects can be composed of 'values'.

For example, all the essentially identical path-operations for spatial objects can now be based on one single value with its operation checking to see whether two objects overlap. Nothing has been changed in the way the path is defined as it still pertains to an object.

Similar to the spatial access we can find many parallels for other access paths where the coding of the equality-criterion is restricted to a single value. Once this operation has been defined it can be used by the all the access paths acting on the same kind of object-part. Direct access to objects, for example, based on a unique integer-number, can be established with the operations related to the integer-type, coded only once.

Values are not integrated into the database, i.e., there is no static dependency between the types available and the database kernel. As values can be added by the user, additional object-types can be provided and the power of the database-kernel is not restricted to a one-time state.

7.2.1 A Minimal Set Of Operations Of Values

The set of operations a value-ADT has to provide to the database is influenced by the kinds of paths in which it is used.

A minimal set of operations consists of routines for
- checking if two instances of the same attribute-type are identical (equality).
- computing the hash value. This routine has to interact with the equality-checking: if two objects are regarded as equal, their hash-values must be the same, too.
- sorting, i.e., checking if one object comes before another one in an ordered sequence.

Apart from the path-based operations, it will be useful to define another set of standard operations which will be needed either in higher layers of the database or in the application programs. Such operations include
- parsing an abstract data type
- converting an abstract data type to an output-abstract data type
- drawing an abstract data type if it is a spatial object
- identifying an abstract data type provided by a 'pick'-operation

8 CONCLUSION

We have presented a data model which is object-oriented and thus suitable for non-standard databases. The data model includes concepts and applications of user-defined abstract data types. This model is based on the abstraction methods of classification, generalization, and aggregation, and includes entities (objects), aggregates, and access paths. It does not include attributes, as they are hidden inside the objects.

Providing abstract data types to the databases implies providing operations. We stated that most of these operations are similar or even identical for different object types. In order not to produce a large amount of code, the operations can be reduced on some parts of the objects we call attributes. Their existence does not change the object-oriented design and interface of the database. Operations included in the abstract data type provide complete freedom to specify how objects are treated. We have shown that a dynamic set of user-defined abstract data types can be combined with the attributes. This extension provides both the necessary test-operations and basic abstract data types which can be used to create higher abstract data types, object-types. They inherit the operations from the lower abstract data types.

PANDA, an existing database, acts along these ideas.

9 REFERENCES

ADA Reference Manual for the ADA Programming Language, Springer-Verlag, New York, 1983

Aronson, P., Morehouse, S., The ARC/INFO Map Library: A Design for a Digital Geographic Database, Auto-Carto VI, 1983

Backus, J., Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs, CACM, Vol. 21, No. 8, Aug. 1978,

Batory, D.S., Buchmann, A.P., Molecular Objects, Abstract Data Types, and Data Models: A Framework, proceedings 10th VLDB conference, 1984, Singapore

Batory, D., GENESIS: A Project to Develop an Extensible Database Management System, International Workshop in Object-Oriented Database Systems, Pacific Grove, California, 1986

Brodie, M.L., et al. (Eds), On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages, Springer Verlag, New York 1984

Brodie, M.L., On the development of data models, in: Brodie, M.L., et al. (Eds), On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages, Springer Verlag, New York 1984

Brodie, M.L., et.al, CAD/CAM Database Management, IEEE Datbase Engineering, VOl. 7, No. 2, June 1984

Buchmann, A.P., de Celis, C.P., An Architecture and Data Model for CAD Databases, 11th VLDB conference, 1985, Stockholm

Cardelli, L., Wegner, P., On Understanding Types, Data Abstraction, and Polymorphism, ACM Computing Surveys, VOl. 17, No. 4, 1985

CODASYL 1971, Database Task Group (DBTG) report, April 1971 (and Journal of Development 1973, 1978)

Codd, E.F., Relational data base: a practical foundation for productivity, Comm. ACM, Vol. 25, No. 2, Feb. 1982

Dittrich, K., Object-Oriented Systems: the Notation and the Issues, International Workshop in Object-Oriented Database Systems, Pacific Grove, California, 1986

Frank, A., Data Structures for Land Information Systems - Semantical, Topological and Spatial Relations in Data of Geo-Sciences, Ph. Thesis, Swiss Federal Institute of Technology, Zurich (Switzerland), 1983

Frank, A., Problems of Realizing LIS: Storage Methods for Space Related Data: The Field Tree, No. 71, Swiss Federal Institute of Technology, Zurich (Switzerland), 1983

Frank, A., Kuhn, W., A Provable Correct Method for the Storage of Geometry, Proceedings of the 2. International Symposium On Spatial Data Handling, Seattle, 1986

Goldberg, A., Robson, D., Smalltalk-80, Addison-Wesley, 1983

Guttag, J., Abstract data types and the development of data structures, Comm. of the ACM, June 1977

Guttman, A., New Features for a Realtional Database System to Support Computer Aided Design, Memorandum No. UCB/ERL M84/52, Electronic Research Laboratory, College of Engineering, University of California, Berkeley, 1984

Haerder, T., Reuter, A., Architecture of Database Systems for non-standard Applications, (in German), in: Database Systems in Office, Engineering, and Science Environment, editor: A.Blaser, P.Pistor, IFB 94, Springer Verlag, 1985

Haerder, T., New Approaches to Object Processing in Engineering Databases, International Workshop in Object-Oriented Database Systems, Pacific Grove, California, 1986

Johnson, H.R., Scheitzer, J.E., Warkentine, E.R., A DBMS Facility for Handling Structured Engineering Entities, Proceedings, ACM Engineering Design Applications, 1983

Knuth, D.E., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading MA, 1973

Lipeck, U.W., Neumann, K., Modelling and Maipulation of Objects in Geoscientific Databases, Proc. 5th International Conference on the Entity-Relationship Approach, Dijon, 1986

Liskov, B., et al., Lecture Notes in Computer Science, CLU Reference Manual, Springer-Verlag, New York, 1981

Lorie, R., Plouffe, W., Complex Objects and Their Use in Design Transactions, Proceedings, ACM Engineering Design Applications, 1983

Maier, D., Why Object-Oriented Databases Can Succeed Where Others Have Failed, International Workshop in Object-Oriented Database Systems, Pacific Grove, California, 1986

Nievergelt, J., et al. The GRID FILE: an Adaptable, Symmetric Multi-Key File Structure, Report No. 46, Institut fuer Informatik, Swiss Federal Institute of Technology, 1981

Olthoff, W., An Overview on ModPascal, SIGPLAN Notices, No. 10, 1985

Parnas, D.L., Share, J.E., Language facilities for supporting the use of data abstraction in the development of software systems, Naval Reasearch Laboratory, Washington, 1978

Scheck, H.-J., Waterfeld, W., A Database Kernel System for Geoscientific Applications, Proc. Second International Symposium on Spatial Data Handling, Seattle, 1986

Shipman, D.W., The Functional Data Model and the Data Language DAPLEX, ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981

Sidle, T.W., Weakness of Commercial Database Management Systems in Engineering Applications, Proc. 17th Design Automation Conf., 1980

Stonebraker, M., Guttman, A., Using a Relational Database Management System for CAD DATA, IEEE Database Engineering, Vol 5. No. 2, June 1982

Stonebraker, M., Rubenstein, R., Guttmann, A., Applications of Abstract Data Types and Abstract Indices to CAD Databases, Proceedings, ACM Engineering Design Applications, 1983

Tamminen, M., Management of Spatially Referenced Data, Report HTKK-TKO-A23, Helsinki University of Technology, Laboratory of Information Processing Science, Helsinki, 1981

Udagawa, Y., Mizoguchi, T., An Extended Relational Database System for Engineering Data Management, IEEE Database Engineering, Vol.7, No.2, June 1984

Wilkins, M.W., Wiederhold, G., Relational and Entity-Relationship Model Databases and VLSI Design, IEEE Database Engineering, Vol.7, No.2, June 1984

Wirth, N., Programming in Modula-2, Springer Verlag, Berlin, 2nd Ed. 1982

Zilles, S.N., Types, algebras and modelling, in: Brodie, M.L., et al. (Eds), On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages, Springer Verlag, New York 1984

PUBLICATIONS
Surveying Engineering Program
University of Maine

The following is a current list of publications by the Surveying Engineering Program at the University of Maine. Copies, unless specifically designated as No Longer in Print (NLP) or available from some other publisher or agency, are available from University of Maine, Surveying Engineering Program, 120 Boardman Hall, Orono, ME 04469. (A fee to cover the costs of processing, printing and mailing is indicated after each available publication.)

7. Land Information Systems for the Twenty-First Century, E. Epstein and W. Chatterton, Real Property, Probate and Trust Journal, American Bar Association, Vol. 15, No. 4, 890-900 (1980). ($5.)

9. Legal Studies for Students of Surveying Engineering, E. Epstein and J. McLaughlin, Proceedings 41st Annual Meeting, American Congress on Surveying and Mapping, Feb. 22-27, 1981, Washington, D.C. ($5.)

15. Storage Methods for Space Related Data: The FIELD TREE, A. Frank in: Spatial Algorithms for Processing Land Data with a Minicomputer, MacDonald Barr (Ed.). Lincoln Institute of Land Policy, 1983. ($15.)

17. Semantische, topologische und raumliche Datenstrukturen in Land-informations-systemen (Semantic, topological and spatial data structures in Land Information Systems) A. Frank and B. Studenman, FIG XVII Congress Sofia, June 1983. Paper 301.1. ($5.)

18. Adjustment Computations, A. Leick, 250 pages. ($25.)

19. Geometric Geodesy, 3D-Geodesy, Conformal Mapping, A. Leick, 420 pages. ($30.)

24. Macrometer Satellite Surveying, A. Leick, ASCE Journal of Surveying Engineering, August, 1984. ($5.)

27. Adjustments with Examples, A. Leick, 450 pages. ($30)

28. Geodetic Programs Library, A. Leick, about 400 pages. ($25.)

30. Macintosh: Rethinking Computer Education for Engineering Students, A. Frank, August, 1984. ($5.)

31. Surveying Engineering at the University of Maine (Towards a Center of Excellence), D. Tyler and E. Epstein, Proceedings MOLDS Session, ACSM Annual Meeting, Washington, March, 1984. ($5.)

32. Innovations in Land Data Systems, D. Tyler, Proceedings Association of State Flood Plain Managers, Annual Meeting, Portland, Maine, June 1984. ($5.)

33. Crustal Warping in Coastal Maine, D. Tyler, et. al., Geology, Vol 12, pp. 677-680, November 1984. ($5.)

50. Instrumentation Needs (GPS and Related Matters), Alfred Leick. Workshop on Fundamental Research Needs in Surveying, Mapping, and Land Information Systems, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, November 18-20, 1985. ($5.)

52. Expert Systems Applied to Problems in Geographic Information Systems, Vincent Robinson, Andrew U. Frank, Matthew Blaze, presented at the ASCE Specialty Conference on Integrated Geographic Information Systems: A Focal Point for Engineering Activities, February 3-5, 1986. ($10.)

53. Integrating Mechanisms for Storage and Retrieval of Data, Research Needs, Andrew U. Frank, challenge paper forWorkshop on Fundamental Research Needs in Surveying, Mapping, and Land Information Systems, November 17-19, 1985, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061. ($10.)

54. Formal Methods for Accurate Definitions of Some Fundamental Terms in Physical Geography; Andrew U. Frank, Bruce Palmer, Vincent B. Robinson. Invited paper at the Second International Symposium on Spatial Data Handling, July 5-10, 1986, at Seattle, WA. ($5.)

55. Cell Graphs: A Provable Correct Method for the Storage of Geometry; Andrew U. Frank and Werner Kuhn. Invited paper at the Second International Symposium on Spatial Data Handling, July 5-10, 1986 at Seattle, WA. ($10.)

56. LOBSTER: Combining Database Management and Artificial Intelligence Techniques to Manage Land Information; Andrew U. Frank. Invited paper No. 301.1 for the XVIII. International Congress of FIG, Toronto, Ontario, Canada, 1986. Commission 3, Land Information Systems. ($5.)

57. PANDA: An Object-Oriented PASCAL Network Database Management System; Andrew U. Frank. ($5.)

58. GPS Network Adjustment with Apriori Information and Orbital Determination Capabilities; Kamil Eren and Alfred Leick. Proceeding of the Fourth International Geodetic Symposium on Satellite Positioning, Austin, Texas, April 28-May 2, 1986. ($5.)

59. MAINEPAC (1): Processing GPS Carrier Phase Observations; Alfred Leick, University of Stuttgart, Federal Republic of Germany. ($10.)

60. Mathematical Models within Geodetic Frame; Alfred Leick, Journal of Surveying Engineering, Vol. 111, NO. 2, August, 1985, (paper No. 19952). ($5.)

61. Deformation Measurements Workshop, Guenter Wittman. ($5.)

63. Three Dimensional Conceptual Modeling of Subsurface Structures, Eric Carlson; Appropriate Conceptual Database schema Designs for Two Dimensional Spatial Structures, Max Egenhofer; both papers were presented at the Eight American Congress of Surveying and Mapping, March 29-April 4, 1987 in Baltimore, Maryland. ($10.)

77. First Steps in Modernizing Local Land Records, Harlan J. Onsrud, Surveying and Mapping, Vol. 45, No. 4, pp. 305-311 (December 1985);
Research for Validating Cadastral Data, Harlan J. Onsrud, Proceedings of the IGIS Conference, Crystal City, Washington, D.C. (November 1987); Technical Standards for Boundary Surveys: Developing a Model Law, Harlan J. Onsrud, Journal of Surveying Engineering, Vol. 113, No. 2, pp. 101-115.

78. The Education of Surveyors and Cartographers in an Information Age, Harlan J. Onsrud, Technical Papers of the XII Surveying Teachers Conference, Madison, Wisconsin (July 1987); Challenge to the Profession: A Formal Legal Education for Surveyors, Surveying and Mapping, Vol. 47, No. 1, pp. 31-36.

79. Data Structures to Organize Spatial Subdivisions, Ian Greasley (Graduate Student in Surveying Engineering) 1987. Paper presented at the ACSM-ASPRS 1988 convention at St. Louis, MO. ($5.)

80. The MainePac Series, Alfred Leick. Paper presented at ACSM-ASPRS Convention, St. Louis, Missouri, 1988. ($5.)

81. Maine Crustal Project: Final Report. Steven R. Lambert and David A. Tyler. Final Report to the Maine Geological Survey on Grant No. NRC-G-04-82-009. 1987. ($10.)

82. Designing a User Interface for a Spatial Information System. Max J. Egenhofer. Paper presented at the ACSM-ASPRS 1988 convention at St. Louis, MO. Partially funded by grants from NSF No. IST-8609123 and Digital Equipment Corporation, principal investigator Andrew U. Frank.
($5.)

83. Graphical Representation of Spatial Data: An Object-Oriented View. Max J. Egenhofer. Paper presented at the third International Conference on Engineering Graphics and Descriptive Geometry at Vienna (Austria). Project was partially funded by grants from NSF No. IST-8609123 and Digital Equipment Corporation, principal investigator Andrew U. Frank. ($5.)

84. A Precompiler For Modular, Transportable Pascal. Max J. Egenhofer and Andrew U. Frank. This paper has been submitted to SIGPLAN Notices. Project was partially funded by grants from NSF No. IST-8609123 and Digital Equipment Corporation. ($5.)

85. Integrating a Database Management System into an Object-Oriented Code Management System. Max J. Egenhofer and Andrew U. Frank. Project was partially funded by grants from NSF No. IST-8609123 and Digital Equipment Corporation. ($5.)

86. Requirements for Database Management Systems for Large Spatial Databases. Andrew U. Frank. Paper presented at the COGEDATA meeting in Dinkelsbuehl, F.R. Germany, December 2, 1986, program for digital mapping in geo-science. ($5)