Data Engineering

Gio Wiederhold and Xiaolei Qian*
Department of Computer Science, Stanford University

1 Introduction

The data-oriented approach to the design of data-processing programs provides a promising alternative to the traditional design of software. We use the term *Data Engineering* to describe this methodology and pose it as an alternative to *Software Engineering*. A number of software engineering techniques remain valid, but the methodological setting is drastically different.

Data engineering focuses on the data. Of prime concern in data engineering are the proper representation and communication of data, so that many processes can share the information contained in the data. This design methodology then depends on an analysis of the meaning and structure of the data. The data engineering approach is appropriate when

1. data *persist* longer than programs; and

2. many programs *share* the same data.

In data processing, and especially in transaction-oriented systems, these conditions dominate.

In data engineering the primitives are structured objects, and relationships among them. In contrast with object-oriented programming, data engineering does not endow the objects with extensive procedural capability, since procedural information is difficult to share, especially in large systems using diverse languages. Instead, we must attempt to represent the semantics structurally, by identifying the structure types that are being used. We associate behavior with these types. For processing the applications interpret the semantics with the typed objects. The concern is not the actual representation; two objects could have identical representation, but be typed distinctly.

Relationships connect the objects. Many processing rules are best derived from constraints associated with connections. Again, one must structurally identify the connection types and permit them to be interpreted.

*Gio Wiederhold is with Department of Computer Science, Stanford University, Stanford, CA 94305; Xiaolei Qian is with Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.

2 Background

Traditional software engineering methodologies focus on the programmed processes of computer applications [23]. Through successive refinement, the application specifications (e.g., data flow diagrams) are eventually transformed into programs, the active part of our software systems. The programs will operate on data, and a concomitant design of data structures is implied (e.g., a data flow diagram is often augmented with a data dictionary), but data are not the primary concern in software engineering. Data engineering presents an orthogonal approach.

Many basic principles of software engineering transfer to data engineering. Data engineering analysis also proceeds *top down*, considering first the meaning of data objects, their relationships to other data elements, and then their proper structure and logical representation. The binding to specific formats, physical data structures, and field sizes is deferred as long as possible. Before actual programs are written, the data specification will have been completed.

In this approach we gain independence of data structures from programs. A variety of programs can be written on these invariant data structures. Not all programs will be written at the same time, since needs and conditions will change.

The long-term goal is that only very high level program specification is needed. Executable code will be generated from the specification using the data descriptions. If the underlying structure changes, recompilation may be needed, but today any but the simplest changes require programmer intervention.

Data engineering has been practiced primarily in database technology. Report generators and some other database tools are examples of higher-level technology. There is no fundamental reason that the techniques used to select and schedule program segments in database management systems should not be applicable to data structures that reside in memory.

The proposed data engineering approach can overcome a cost barrier hindering acceptance of automatic programming. For an executable program both data structures and programs must be generated. Lack of reuse of data structures means that the full semantics must be specified for every program to be generated. Unfortunately, the cost of specification is comparable to the cost of procedural programming. If the portion of the semantics assignable to the data structures does not have to be repeated, automatic programming will be enhanced.

We find here also an alternative to program reuse. It has been very hard to reuse program modules where the semantics could not be stated in a few sentences. Trigonometric functions, search and sort routines, matrix manipulation routines have yielded to reuse, but programs that required more complex data structures have rarely been reusable.

3 Motivation

Problems encountered in data-processing provide incentives for the data engineering approach, but the issues occur in all kinds of computations. In data-processing many small programs share access to the same data. Some of these programs will be used to retrieve information, others are used to update the data, so retrieved results will be up-to-date. A company, say in the business of leasing movie films to theaters, has to keep track of its own movie inventory and about new films coming out. All these transaction programs are simple and operate on a shared data structure. Other programs are run periodically to produce reports. Such a program may track the popularity of each film to schedule future distribution. Competitive analysis can compare the status of the company with others in the field. Only a few of these programs are initially defined. Large analysis programs may not be productive until a few years' worth of data has been accumulated, and yet, the data structure chosen must accommodate any of these subsequent tasks. After a few years the original programs may no longer be in use, but the data structures persist.

The motivation for a data engineering approach is not limited to small programs dealing with voluminous data. When a large single program has been decomposed, recursively, into many small modules, many of these modules will share information. These modules are likely to be implemented by distinct programmers and certainly by distinct programs. These programs, if they do not handle compatible data structures, must transform the shared information to their own structural specification. When they are used sequentially, the modules may *pipe* data to each other, and during piping may carry out such transformations. When data are voluminous, such transformations are costly.

If the modules are used cyclically, then the transformation of data also cycles among the programs' data structures. The computational cost of repeated transformations is high and information may be lost in the process. Information for some tasks may be implicit in some data representation, say in the order that similar tasks to be scheduled are ranked. Such information is lost if only the explicit task attributes are used in the transformation.

A third motivation for data-oriented design stems from the need to communicate data. In modern computer systems, many computing nodes participate in data processing. The representation and validity of similar data elements will differ among nodes. Some nodes will maintain data derived from remote sources. In order to access and share such data values the shared data must be well-described. Only then will users in different environments be able to work together.

4 Method

The Data Engineering approach starts by considering the information structure of the domain to be covered. The information itself may be labeled either data or knowledge. We use the term *data* for the major structures which describe the state of the world within the application domain, and the term *knowledge* for the metadata, description of those data.

The design process, since it most applies to data that are constantly changing as the application world changes, focuses on establishing the metadata. The underlying assumption is that the data are voluminous and rapidly changing, and that the metadata is of manageable size and does not change rapidly.

Eventually applications will have to process data. The application is seen as a transformation specification, which use the metadata to determine how to obtain the needed data, which transforms can be carried out, and how to best carry them out.

The principal tools serving the data engineering approach are

1. the *data dictionary*, describing all data elements of concern; and

2. the *data model*, describing the relationships among data elements.

To some extent the data dictionary is oriented toward the syntax of the data, and will contain a great deal of detail. With the name of an element the data dictionary will list the format, any subcomponents, the size, etc., of the element. During the design process the entries are continuously expanded and refined.

The data model captures the semantics, and hence the meaning of the data. A data element, say the `number_of_performances`, means little until we know the title of the movie, where it has played, and the size of theater. For analysis we will also want to know about the director of the film, the star, the public attending that theater, etc. Data and people are made relevant by their participation in relationships, and the recording of such relationships is an essential focus of data engineering.

5 Data Dictionary

Programmers use the data dictionary as the prime documentation for writing the application programs. It is primarily a table which lists, for each data element, the information needed to assure its correct use. Many types of information are collected there, coming from a variety of sources. We present the categories of information in Figure 1; much more structure could be incorporated, but we leave that to the subsequent discussion.

A data dictionary is incrementally updated as the data are defined, and as other users and programs start using these data. We will briefly discuss the various categories of information found in a data dictionary.

5.1 Names

One name must serve as the identifier. The identifier must either be unique or always permit disambiguation. Names may have a qualification attached to them to assure uniqueness in context, e.g., `Movies.title` versus `Books.title`. While a single common name is the best basis for communication among programmers, at times restrictions or conventions force the use of alternate names. For instance, a programming language with a seven-character limitation for variable names may force the element name "`director`" to be abbreviated. The choice of `directo` versus `directr` must be documented. The full name may include a more precise specification: "`Primary director, at the time the film was completed`".

Since a data dictionary is developed within a specific context, it is rare that all names are properly disambiguated. In the `Movies` environment `actor` needs little disambiguation,

Data Attribute[a]	Definition and Usage
Identifier	Short name, as used in programs
Alternate identifiers*	Any other names used in programs
Full name	A complete descriptive name to avoid ambiguity
Domain	Abstract data type pertaining to instances referred to by the identifier
Part of	Member of higher level abstract data type
Composed of	Nested data elements included
Range	Limits, or a list of permissible values
Representation	Format of data values, i.e., `REAL`, `CHAR`, etc.
Size	Number of bits or characters, or limit, if variable
Marking	Method of denoting variable length (count, terminator, ...)
Cardinality	Expected number of elements
Specifier	Designer of this data element specification
Owner	Individual responsible for data values
(Accessors: access types+)*	Users having access privileges
Sources+	Names of programs which create new data values
Modifiers*	Names of programs which modify the data values
Uses+	Names of programs using the data
Load estimates*	Frequencies of update and retrieval
Storage node and file	Persistent storage for the data values
Backup files*	Storage nodes and files for recovery of the data
Secondary files*	Other files containing the data
Comments	For anything not categorized above

[a] Attributes marked * have zero, one, or more entries. Attributes marked + have one or more entries.

Figure 1: Contents of a Comprehensive Data Dictionary

although the fact that `actor` is also a term in a software methodology is unrecognized. More subtle differences in scope are even harder to deal with. The movie industry may not recognize every person who appears in a TV commercial as an `actor`. The difference can become crucial when payment for residuals is being computed.

5.2 Domain and Format Specification

Not only the eventual representation and size decided on for each data element are recorded. The domain specification provides the abstraction which leads to the chosen representation. In the representation, size may be made larger than necessary for alignment reasons on some computers. When data are placed on other computers, the domain information should be consulted, since alignment rules may be different.

Primitive domain types are predefined:

1. Integers, e.g., `316`
2. Reals, e.g., `6.2`
3. Characters or character strings, e.g., `"Bogart"`
4. Boolean, e.g., `True`
5. Date and time, e.g., `10Sep1988, 14:09`
6. Identifier used for reference, e.g., `Actors.Bogart`

The two top items in the list have well-defined computer representations. The other domain types listed can have a variety of representations. Character strings and their terminators vary somewhat among software systems, even on identical computers. Representations for boolean and `date_time` vary greatly. Some representations for `date_time` restrict the domain. A referencing identifier must be unique, or at least resolvable to a unique element in any given context. The context may be attached as a qualification prefix.

Further domains may be created and used as needed, as `dollar_amounts` as a specialization from the primitive domain `real`, or `movie` as a specialization of `identifier`. New domains can be built by combining other domains, say `performance` from `movie` and `date_time`.

The actual data representations required to represent these domains can be defined late in the software design process. They may be primitive or may be structures, such as lists or arrays of other types. A frequent, yet simple, data type is a reference which cites some other data structure by its identifier. All structures used to create complex domains will be specifically defined in the data model as well, since they represent important relationship semantics.

Derived data types are those that are computed internally, rather than entered as input. For these derivations, source data types should be noted.

The values that the data elements may take on will further determine the representations to be implemented. For instance, given that for `year_produced`, we applied a specialization and truncation of the basic type `date_time` having a `range`: {1896,2000}, we find that 105 values or 7 bits suffice. For the set of display colors, `Red, Green, Blue`, three values suffice; for the set of studios making films, the domain may be defined by reference to a file containing studio names. `Studios.name` is again a specialization of the primitive type `identifier`.

Information on cardinality of occurrence is relevant if the element can appear repeatedly. Cardinality is typically only an estimate. A range is often the best way to specify it, so that the reasonableness of the bounds can be discussed.

5.3 Accessors

Many people participate in the design and use of data structures. To assign authority and responsibility we, first of all, name the creator or maintainer of the specification. The actual authority over the values is assigned to the *owner*, someone involved in the operation of the enterprise. The owner may allocate further access rights to others. When a person is given authority over data, that person also assumes obligations to maintain the data correctly and to keep them up-to-date. Classifications of authority are composed of combinations of basic privileges: `READ, EXECUTE, CHANGE, DELETE, EXTEND, MOVE, EXISTENCE VERIFICATION`. Not all of these types or their combinations are directly supported in today's systems; some adaptations will be made during implementation. `EXISTENCE VERIFICATION`, if disallowed, implements hiding.

Often the delegation of authority is to programs, rather than to people. The programs in turn are accessible according to privileges granted to people. The intermediary programs, however, may be verified to help assure that the data are not modified inappropriately. The data dictionary and the data model will define the constraints to be applied to the programs. Estimates of access load help in making mapping decisions during implementation, as described in Section 7.

5.4 Files

Data which persist beyond the lifetime of a program are stored within files on computers. With each file we identify a node name rather than a specific computer, since clustered computers within a node may be sharing files. A database management system may maintain some files, giving a greater level of automation to the control functions. Often multiple depositories must be noted, since computations can copy and transform data, or ship data to other, perhaps remote computers. It is common that one depository is defined as the primary one. In case of failure others may be used as backup. Secondary copies should be updated as needed, to maintain global consistency.

5.5 Software for Data Dictionary

The use of data dictionaries has become common in large data-processing organizations. A number of software packages are available to serve the need, but many are associated with specific database management systems. They rarely provide higher level domain abstractions because domain knowledge is a higher level of abstraction then is provided by current systems. The domain types that one can specify are typically limited to the representations supported by these systems. Furthermore, their built-in editors are often horrid. Because data dictionary tools available today are limited, programmers often extend or build their own data dictionaries using text editors and their own conventions. Some large user organizations have standardized their content for internal use. Federal and international efforts at standardization are also in progress [28, 22].

A data dictionary is often extensively used in design, but when the program product is turned over to a user group, it may not receive much attention. Programmers and designers like to believe that their work is permanent and without fault. As a result, the data dictionary is often not well-maintained. Pressures to get programs extended and debugged often leave too little time to update the data dictionary with the results of program additions and changes. An obsolete data dictionary can of course cause further problems.

6 Data Model

The data model defines the relationships among the data elements. The data model and the data dictionary are defined concurrently during the design of the persistent data structures. The data model cites the data dictionary frequently. Little detailed information will be available in the data dictionary at early stages in the design, one should only insist that an entry exists for every citation made.

The literature on data models focuses on database concepts. Relationship encoding and structural constructs as hierarchies and networks are crucial. Early broad examples relevant to our concerns are [1, 21, 32]. Those concepts are typically associated with specific types of database management systems. To enable effective interaction with their users, the data models used in databases must capture some of the semantics of the applications. Research in Artificial Intelligence is contributing to this field [5, 19]. The associated models range from fairly simple and formal ones, as the *relational model*, to quite general ones, such as *semantic nets*. We also see specialized models generating much interest, e.g., in the design of products ([4]) or in law ([33]). Rather than surveying the field, we use here one specific model, the structural model, which falls between the two extremes and is not bound to any specific system type.

6.1 Basic Concepts for Data Modeling

The possible range of semantic relationships is without bounds, since all kinds of semantics are expressible by the data structures we can create. We must reduce the types to a workable number, and we do that by focusing on the structural alternatives, rather than on the expressible semantics.

To make the model workable, we first limit ourselves to binary relationships, i.e., relationships among two data element types. By having an adequate algebra we can group elements and compose basic relationships into ternary or more complex structures. We also limit the model to relationships which can be directly implemented without hidden data structures. This means primarily that we will not represent $m : n$ relationships, which require an intermediate data structure for their interpretation. This relationship type is to be composed.

Relationships are associated with *dependencies*. If we talk about a `director`'s work we better know the `movie`. Only if we have a unique definition of the `movie` can we know which specific director is meant. ¿From knowing the `director` we cannot know what `movie` is meant — unless the `director`'s career was quite unsuccessful and limited to one film. There is hence a directed functional dependency from `movie` to `director`. The concept of dependencies is a primary basis for data modeling.

We will also encounter other dependencies in modeling. An *inclusion dependency* defines the case where a nest of several data elements together depends on a single element, as the actors of a film.

For each data element defined in the data dictionary we analyze its dependencies on other data elements. Dependencies imply that the dependent value must change when the ruling value changes.

6.2 Primitives of the Structural Data Model

Only seven basic primitives are used in the structural model presented here. Enforcing a limit on the number of primitives helps in education and application; however as mentioned above, several primitives must be composed to describe complex structures [25].

1. Aggregation: bringing together of dependent elements into records
2. Collection: grouping of repeated elements into arrays
3. Nesting: a connection to an included collection forming a level of a hierarchy
4. Reference: a connection constraining a remote aggregation
5. Generalization: bringing together of similar aggregations
6. Derivation: defining value dependencies
7. Identity: defining permitted asynchrony

These primitives, albeit under a variety of names, also occur in other modeling systems, but for consistency we will define them here independently.

Some of the primary references for each concept, as used in data structuring, are given here, although all can be related to prior mathematical concepts. Most of these concepts were already used in the late 1960s [34] to describe persistent structures, but the model did not include rules to permit their use in a prescriptive manner. Those models appeared subsequently. Collection and aggregation are basic to the relational model [12], and later constructive rules were associated with that model. The Entity-Relationship (E-R) model gave a major impetus to semantic modeling; the associated graphics proved to be an effective communication tool among users and designers [8]. The semantic knowledge obtained in E-R modeling provides informally some prescriptive guidance in aggregation, collection, and referencing. Aggregation and generalization were elucidated by Smith and Smith [37]. El-Masri [17] extended the E-R model with categories. The resulting E-C-R model can model generalization. Nesting was an early concept, used already in 1959 by Prywes and Gray [29] and modeled by Bachman in the work leading to early database systems [2]. Nesting, reference, and generalization are the concepts used to extend the relational model into a structural model adequate for the formalization of E-R and E-C-R models [40]. Identity is a more recent concept, it is required when modeling relationships among distributed data on autonomously operating computers [41].

6.2.1 Aggregation

To permit joint management of several elements which depend on the same *key* element, we can group such elements into keyed records. We extend the concept of record beyond the common data structure [44] by identifying one or more key elements which rule the remaining elements. The ruling elements appear first in the list, separated by the symbol :⟩ and the entire list is placed in curly brackets ({ list }). An aggregation has then the format

name : { ruling_part :⟩ dependent_part }

where conditions of Boyce-Codd Normal form hold:

Each dependent element depends on all ruling elements.

No other dependencies are permitted [13].

In our examples we can define

`Movies:` { `movie` :⟩ `director, studio, year_made, ...`}

If the problem semantics cannot assure that a unique ruling part can be constructed from the given data elements, then it is necessary to insert a *surrogate* data element of type `identifier` as the ruling part [14]. We note, for instance, that there are 5 movies with the name "Frankenstein". Then we might better model

`Movies′:` { `movie_id` :⟩ `movie_title, director, studio, year_made, ...`
}

The actual database user cannot be expected to know the `movie_id`. To determine a specific `movie_id` the `year_made` or `director` may be needed in addition to the `movie_title`; in some cases even more element values may be needed.

In an object-oriented programming paradigm, such *object-ids* are considered intrinsic and often invisible to the user. That approach helps in system management, but the model semantics are not greatly affected.

6.2.2 Collection

Records, as defined by aggregation, may repeat. The aggregation concept does not permit any repetition within its data elements; all its elements are simple. Defining a collection permits the construction of a file of records; all the records in a collection are similar. If not all instances of an aggregate are to be included in a collection, a predicate expression can be appended to provide inclusion or exclusion conditions.

The collection is named so that it can be treated as a single object. Associated with a collection are functions which permit the extraction or removal of a specific element from the collection. We denote collection simply by naming bracketing:

```
collection { aggregation | predicate }
```

so that we might say

```
Stars { Actors | fee > $1,000,000}
```

The constraints imposed on collections mean that the individual elements need not be named here, and that uniqueness is assured.

Up to now we have remained within the concepts derived from the relational model. The application of these two modeling concepts permits construction of many conceptual structures out of the defined data elements. To serve data-engineering we have to extend these concepts to permit capture of further semantics with our structural model.

6.2.3 Nesting

Collections of elements or aggregations can occur within other data structures; we refer to such subsidiary collections as *nests*. A nest forms one layer of a hierarchy. Ruling parts are also needed to properly identify nested values. We will always identify the nested values with an inherited ruling part, taken from the collection which contains the nests. The general notation is then

```
Collection.ruling_part ——* (Collection, Nest).ruling_part
```

and the `Nest` collection will contain further ruling and optional dependent elements.

If we define inside a record for a `movie` a data element for `actors` we need a structured element which can hold multiple simple entries for the cast of actors. In our case we have

```
Movies: { title :⟩ director, studio, year_made, ..};
Casts: { title, actor :⟩ role};
Movies.title ——* Casts.title;
```

`Casts` has instances as

```
Casablanca, Bogart :⟩ lover;
Casablanca, Dooleg :⟩ pianoplayer;
Casablanca, Bergman :⟩ beloved; ...
```

A citation from the model record for the `Movies` to the model record for the `Casts` defines the nesting of the subsidiary collection. We repeat the ruling part, `title`, from the model record `Movies` to identify instances of records described by the model record `Casts`, since these records have to serve many `Movies`. Individual `actor` names will also appear multiple times in that set of records, the multiplicity is symbolized by the $*$ at the destination of the ——* symbol.

The instance of a citation is more than a simple reference, it refers to a nest having anywhere from zero to many subsidiary, *owned*, records. The semantics of the *ownership* connection are that instances of subsidiary records may only exist as long as the corresponding superior, *owner*, record exists. The nested instances hence have the same operational semantics as a single dependent element, say the entry `year_made`.

The model permits denotation of this connection at a higher level of abstraction. We use the notation `Movies.title ——* Casts.title` to indicate that any instance within the collection `Movies` can own instances of `Casts`. The data element names of the aggregations (`title`, $2\times$), attached to the collection names (`Movies, Casts`), are actually redundant. Inspection of the model records reveals unambiguously the details of the connection. The values in the record instances also reveal unambiguously which instance records are connected: The values in the `title` elements must match.

Nesting can be applied recursively. During the production of a film there may be a list of the scenes in which each actor appears, modeled as

```
Scenes: { title, actor, scene :⟩ }.
```

An instance would be

```
Casablanca, Bogart, 17 :⟩ .
```

6.2.4 Reference

While nesting permits the construction of a hierarchical data structure, hierarchies by themselves are inadequate to deal with general relationships among objects. We encountered the concept of a reference already when discussing data types. We now expand the semantics associated with that concept, in order to assure that we maintain correct data structures.

Defining a reference to a remote object places a constraint on that object: it *must* exist. A program may attempt to reference objects of unknown state as long as a test is included to check that the objects are valid; but much programming of data structures proceeds with the assumption that the data elements exist. That semantics we capture here formally.

We denote a reference as follows:

`referent.elements` $\succ\!\!\!-\!\!\!-\!\!\!-$ `referencee.ruling_part`.

For instance

`Casts.actor` $\succ\!\!\!-\!\!\!-\!\!\!-$ `Actors.name`

given as referenced object another collection of aggregations containing relatable data

`Actors:` $\{$ `name` $:\rangle$ `type, star_rank, fee, works_in, ...` $\}$.

The Destination of a reference is always the ruling part of an aggregation but any data element may be of type referent. For instance, `works_in` may in turn be a referent to a collection `Countries`:

`Actors.works_in` $\succ\!\!\!-\!\!\!-\!\!\!-$ `Countries.name`

where

`Countries:` $\{$ `name` $:\rangle$ `continent, population, film_production`$\}$.

References may originate from multiple collections. For instance `Countries` is also referenced from the `Studios` collection.

`Studios:` $\{$ `name` $:\rangle$ `city, country, film_production`$\}$

and the model states

`Studios.country` $\succ\!\!\!-\!\!\!-\!\!\!-$ `Countries.name`

The referenced aggregations are constrained. The rules for reference management are that the referenced object must exist before the referent is inserted, and that the referenced object may not be deleted while any reference to it exists. The data model provides the documentation for these rules, leading to correct maintenance of the data structures. Note that, in the data engineering approach, the definition of these rules precedes, and if necessary supersedes, any definition implied or implemented by a program.

Since the referent and the ruling part of the referenced object are identical, it is possible to replace the reference with a pointer. Pointers can increase the speed of access but reduce flexibility. But decisions on such tradeoffs are best deferred to the implementation phase.

6.2.5 Generalization

Our models, and the generated programs, increase in power when one abstraction can cover a greater collection of instances. For instance, both `Directors` and `Actors` are instances of people, and some `Actors` have in fact become `Directors` and vice versa. Defining `People` as an entity will permit reuse of concepts and increase consistency.

Some data elements are best associated with generalization `People`, say `date_of_birth`, and others with the specific subsets, as the `star_rank` of the `Actors`. The specialization avoids redundancy or ambiguity in data entry; only for `Actors` will the rank be requested. During analysis it avoids errors, the subsets for a request, say `Average(fee)`, are precisely defined.

To denote generalization we define a connection from the generalization to each object specialization:

```
General.ruling_part ——⊃ Specific.ruling_part
```

The ruling parts, since they identify the same object, are always identical.

Extending our previous example with the generalization

```
People:  { name :⟩ gender, date_of_birth, nationality, ...  }
```

permits the definition of connections to the subsets

```
People ——⊃ Directors
People ——⊃ Actors
```

For `Directors` and `Actors` we keep specific data. The collection of `People` can of course also include other types of folk, and if nothing special is to be aggregated for them, no specific subset is required.

However, any instance in a subset must also appear in the generalization. The data elements in the subset are likely to be incomplete for many computations. The deletion of a general instance then forces the deletion of a subset instance. Subset instances can be freely deleted, as long as no other conditions are violated. A person who is fired as a director remains a member of the `People` collection. In our current model we could not, of course, remove an `Actor` instance, as long as the actor appears in the collection `Casts`, and hence could not remove the corresponding `People` instance either.

Control is brought to specializations by specifying data elements which define subset membership explicitly. In our example we would have

```
People:  { name :⟩ is_dir, is_act, ..., gender, ...  }
```

and we add a predicate to the definition of the connections:

```
People ——⊃ Directors | is_dir
People ——⊃ Actors | is_act
```

If we wish to disallow that directors also become actors, we can add further predicates:

```
People ——⊇ Actors | is_act ∧ ¬is_dir
```

This predicate will not inhibit actors from becoming directors, but then they cannot remain actors. There is only a limited concept of ordering events by time in the structural model, introduced in the next section.

There remains an important case where the generalization is such that all subsets are disjoint and complete. This specification is global over all the connections and is associated with the general collection itself:

```
People:  { name :⟩ one_of(is_dir, is_act, ...), gender, ...  }
```

Such a specification assures us, for instance, that once we have dealt with all subsets of `People`, we have dealt with all `People`, but only once with any individual. The corporate `payroll_department` appreciates such an assurance.

6.2.6 Derivation

Not all data stored in our computer systems are obtained by data entry. Computations contribute *derived data*. Derived data are, strictly speaking, redundant. As long as we have access to the computational algorithm and the source data, they can be recomputed. Storing derived data is beneficial since it avoids the effort of repeated recomputation and excessive access to past, historical data.

For persistent derived data we must include, best in the data dictionary, the source of data. We exclude in our model the possibility that a single data element may be either (1) derived, or (2) entered directly by a user. Such a mixed case is handled indirectly by a structure which uses distinct data element entries for the derived value, for the input value, and, if necessary, for the value selected to be used under specified conditions.

For derivation we invert the common notation: `result := function(input_variables*);` to simplify its incorporation into other constructs. The functional form we use for our programming environment states

```
Program_name(arguments*) =:  result.
```

All of the `arguments` are considered to be *value* or input variables, so that we implicitly disallow side effects in the model. While side effects will occur within programs, their use should be explicit at the level of abstraction dealing with persistent data. The `result` can be an aggregation which explicitly accounts for all effects.

In our example setting we might have

```
Income_estimate (Theaters.type, Theaters.size, Movies.rating) =:
Theaters.income
```

Simple computations as shown above are regarded as atomic *transactions*, so that we are not concerned in the model with specification of the internal behavior of a transaction [39]. One expects that the transaction program will be correct, and when it terminates, satisfy all correctness conditions imposed by the model. Software and data engineering tools, applied during the software generating phase, help assure that this expectation is fulfilled.

6.2.7 Identity

The six previous concepts (aggregation, collection, nesting, reference, generalization, and derivation) are defined without the notion that a delay might occur in establishing the specified relationships and results. Large programs cannot be expected to carry out all computations synchronously. This condition is exacerbated when multiple computers and networks are involved. Traditionally these issues have been left to programmers, although recently some researchers have been addressing these issues [20]. While we do not have general algorithms to manage data without insisting on strict synchrony, we do need specifications to permit more relaxed requirements to be imposed.

The principle is to specify the permitted asynchrony. The model denotes this by connecting two data elements, which should *eventually* be identical, with an *identity* connection. Associated with an identity connection are the conditions when identity must strictly hold.

```
source —= destination | condition
```

We recognize three types of conditions:

Δt	The identity must hold after a time interval `t`.
@T	The identity must hold at time `T`.
!P	The identity must hold prior to execution of procedure P.

Examples from our domain might be

```
Studios.hours_worked —= Payroll.hours_to_be_paid | Δ15min.
Studios.work_schedule —= Front_gate.schedule | @7:30am
Theaters.income —= Properties.income | !Profit_calculation
```

The procedure identified by the "!" symbol defines a derivation, and one would expect the destination to be properly computed if the identity expression is an argument to that procedure.

The source data element may also be an expression. In that case the previous derivation can be merged:

```
Income_estimate(Theaters.type, Theaters.size, Movies.rating) =:
        —= Properties.income | !Profit_calculation
```

Often the destination nodes only require selected or summary information:

- A particular `theater` only needs to receive data about its own future schedule.
- The central `payroll` office will only receive total hours worked ¿From the `studio` node, not the detailed time records.

It is easy to envisage the need for specification of time constraints among distinct computing nodes. A primary function of information systems is to combine data from multiple sources. It is difficult to synchronize multiple local computing processes and rules globally

Model Primitive	Cardinality Source:Destination	Symbol	Instance Update Rule Instance Deletion Rule
Aggregation	$(k) : d$	$:\rangle$	Ruling part (key) is unique No direct constraints
Collection	$1 : c$	$\{\,\}$	Aggregation type must match No direct constraints
Nesting	$1 : m$	$\longrightarrow\!\!*$	Source (owner) must exist before Destination (nest) can exist Source deletion implies Destination deletion
Reference	$n : 1$	$\succ\!\!\longrightarrow$	Destination (referencee) must exist before Source (referent) can exist Destination cannot be deleted when Source exists
Generalization	$1 : 1$ partial	$\longrightarrow\!\!\supset$	Source (generalization) must exist before Destination (specialization) can exist Source deletion implies Destination deletion
Derivation	$a : 1$	$=:$	Source (argument) update implies Destination (result) update Source deletion implies Destination update
Identity	$1 : 1$ delayed	$\longrightarrow\!\!=$	Source (primary) tuple is updated first Source is deleted first

Figure 2: A Summary of Data Modeling Concepts

[7]. Formally results will only be valid if all data are properly synchronized. Although the absence of a specification implies that all data will be available when needed, such an ideal is easily unwittingly violated.

For data models, the identity connection, with its meaning of *eventual* synchronization, means a relaxation of implied requirements. The conventional database processing paradigm assumes that all work is completed within atomic transactions, although complex tasks may require nesting of transactions [26]. Today, timing requirements are either unavailable or specified informally. Only in so-called real-time systems do we find the kind of specification formalized above. For programmers, having an identity specification forces more precision on their implementations.

Dealing in more detail with the semantics of time remains for now a task of the process definitions. Progress can be expected; research into temporal models is an active area of research [11, 18, 27, 38].

6.2.8 Summary of the model

We now show all seven primitives in a concise way in Figure 2. With the primitives we state the constraints which are needed to maintain structural integrity. These were only informally discussed up to now.

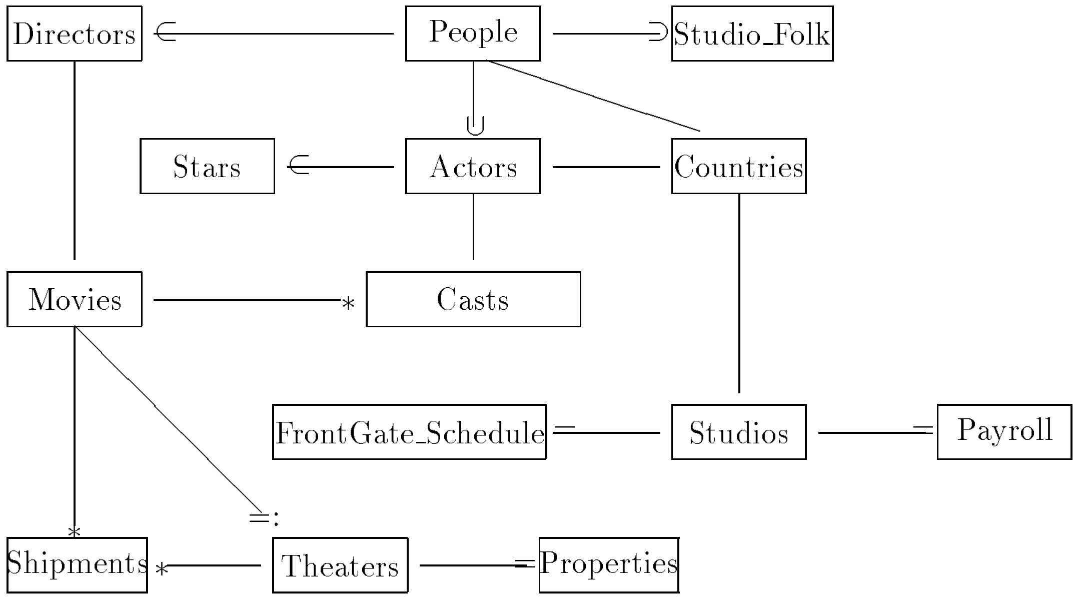

Figure 3: Data Model for a Movie Leasing Company

6.2.9 A Complete Example

Figure 3 exposes the complete sample model we have built up thus far, with some augmentations. We include collections for `Directors` and `Studio_Folk` in addition to `Actors`, and limit `People` to these three categories. A reference from `People.nationality` to `Countries` seems also desirable. By adding a reference connection from `Movies` to `Directors`, we inhibit removing instances of directors as well. Other `Studio_Folk` are not that lucky, and can be freely removed. `Shipments` give the shipping schedule of `Movies` to `Theaters`. To make the model diagram more readable, we omit the data elements and only show dependencies between model records.

6.3 Tools for Data Modeling

Tool development for data modeling is relatively recent. Some early development was initiated by projects focusing on data translation [36], but tools developed there remained at a low level of abstraction. The success of the entity-relationship model has encouraged the development of design tools at that level [9]. Knowledge-based systems for interactive design focus on data structures [43]. Languages used for AI have been used to model abstractions covering many instances, for example PROLOG [33] and an object-oriented derivation, OBLOG [35]. These tools are not yet coupled with the data dictionary, so that the concurrent development of representation knowledge, leading to implementable data structures, is not well supported.

7 Data Structure

The model concepts developed in the previous section provide the basis for the selection of appropriate data structures. The constraints associated with the concepts determine the programs which interpret them. A variety of data structures satisfies stated constraints because any data structure is augmented by programming. The primary criterion for the selection is correctness. When alternative data structure representations are possible, the ease of interpretation provides a secondary decision criterion.

7.1 Correctness, Clarity, and Safety

Correctness in small structures and code sequences is often obvious, but it is awkward to provide rigid rules, since exceptions are always possible. For instance, aggregation is conveniently implemented as a record structure. Using an array to hold the sequence of elements is not wise, because the elements are not constrained to be homogeneous. However, an array may do well in an application where all record values are numeric.

If the structure is difficult to interpret, the programs will take more time to execute. For instance, to assure that all references are valid for a new record entry requires finding the matching ruling part in the referenced collection. We want to make sure that the entries in the collection are easily located. Here fixed-size entries are best, otherwise auxiliary structures are needed. For example, a lot of storage space is wasted if we require `movie` to be fixed size. Therefore it is better to have a fixed-size `movie_id` as the ruling part of `Movies`, and an auxiliary table establishing the correspondence between `movie_id` and `movie`.

Another criterion is safety: if a program or the computer fails, will the damage be serious? Typically, structures which depend on pointers are more prone to failures than structures using tables and manipulable identifiers. Specifically, if the entries in references are implemented by pointer chains, an entry being switched may get lost if a failure occurs in the process.

7.2 Performance

Computer scientists have analyzed the relative algorithmic performance of individual data structures. The performance of an entire system will depend on the total usage of each structure. The data model can help in documenting usage information such that expected loads of information retrieval and update on the system can be estimated. Two types of usage may be documented:

Frequency of use of a data element or data element type The primary supporting data structures inherit this access frequency derived from system usage estimates. In our sample company we know that the `hours_worked` in the `Payroll` department is updated daily, and then processed weekly, quarterly, and yearly. Subsequently the data is archived. There may also be some ad hoc requests for the value of `hours_worked`. This information provides an estimate of access frequency not only for the element `hours_worked`, but also for the aggregation: the record for that `Person`, and the collection: the `People` file.

Frequency of path computation The major cost in data processing is not computing on the data, but locating the data. This cost depends on the data structures chosen. If the data structure is a list, we must traverse the structure; if it is an array, we may be able to use a subscript; if it is a record with fixed entries, we find the element via an index to a table of displacements; while a record with variable length entries may require scanning and counting end-of-string markers. The expected frequency of access operations affects greatly the importance of making wise choices of data structures. To update `Bogart`'s individual record rapidly in a large collection we may want to use a hash function and implement the collection as a hash table. However, now the weekly processing becomes awkward, if the collection is then processed sequentially by `name` or `number`.

The data dictionary permits documentation of the estimates. The data model permits determination of the access paths so that estimates can be assigned and inherited.

Comprehensive optimization of structure choices is rarely worthwhile; there is too little of precise knowledge of usage. But since the choices of available structures are limited, the estimates provide invaluable guidance when mapping model concepts to actual data structures. It is certainly not worthwhile to create complex structures to support infrequently-used paths among the data elements.

The continued availability of the model also helps when initial usage differs ¿From expectations and when over time usage changes. The effects of suggested implementations can be determined and their secondary effects can be estimated.

7.3 Implementing Data Structures

The choice of programming languages and systems will also have an effect. Object-oriented programming environments provide a greater clustering of data structures. Their interpreters also permit great flexibility in implementation [15]. Databases fix the structures largely independent of the interpreters, and here early implementation decisions are hard to correct. A major task becomes the optimization of database query execution. Such optimizations are specified by declarative languages. General, procedural, programs cannot be easily optimized.

In Figure 4 we list data structures and mechanisms which are likely appropriate to represent the concepts presented above.

The choice of data structures requires care. For persistent information the selected data structures will be more permanent than the program structures. New programs may be developed to use the data, and earlier programs may fall into disuse, but the organization of the data remains.

Data engineering encourages the modularization of programming tasks into small units or transactions. Having well-documented data structures encourages the sharing of data.

Concept	Representations
Aggregation	records, heterogeneous lists, catenation of strings
Collection	arrays, homogeneous lists, pointer lists, files
Nesting	groups, recursive lists, trees, owner-member links
Reference	identifiers, pointers, hashing, unification, referential integrity
Generalization	trees, rings of pointer, multi-lists, type hierarchy
Derivation	functions, programs, transactions, inference rules
Identity	triggered programs, scheduled programs, blackboards

Figure 4: Candidate Data Structures

Sharing, in turn, encourages the modularization of software into transactions, which operate independently subject to the modeled constraints. Transaction processing simplifies the programming effort; the data becomes the medium which links all activities together.

Operating systems are starting to support the transaction concept directly, so it is no longer necessary to operate in a database system to exploit the concept. Some of the motivation for operating system developers is the need to deal with an increasing distribution of tasks over systems having multiple computing nodes [10]. Here also programs must be decomposed into independently operating units.

7.4 Data Engineering and Data Communication

The spread of distributed computing emphasizes data issues. Shipping of information across communication links is costly relative to local computation. Requirements to transmit either programs or data encourage modularization to gain performance.

We expect that

1. Data are frequently updated, since they should describe the current state of the world.
2. Data descriptions, as derived from the data dictionary and the data model, will not change as often. Most changes will also require much change to the stored data.
3. Existing programs depend on the data structures as described in the data dictionary and the data model. They will also not change often.
4. New programs can be inserted at any time, unless they alter data in a way that affects modeled constraints.

In order to process data the processing programs must eventually be located where the data are.

We can now consider issues of program shipping versus data shipping. Note that the relationships between programs and data are of cardinality $p : d$. Any program may use several distinct data structures and one data structure may be used by a number of distinct programs.

7.4.1 Sharing Programs Remotely

Existing, stable programs are best located where the data are. Programs, especially if they are of modest size, are easily replicated at the nodes holding the data. Data can also be replicated, but replication of data which change frequently is costly. Such data may be fragmented into small data structures, so that the most volatile fragments can be isolated.

Programs dealing with volatile data are best placed at the primary update node. Data and data fragments that change infrequently can be replicated for use at secondary nodes. Replication reduces access cost by programs which are bound to reside at nodes where their volatile data are.

Modularization of programs, that is, executing code as much as possible where the data are, reduces replication requirements. Fragmentation of data makes shipping for replication and processing less costly [6, 31].

7.4.2 Sharing Data Remotely

In order to fragment, locate, and ship data over communication links we need formal data descriptions. Locating the appropriate data requires sharing access to the data dictionary. Sharing requires consistent formatting and encoding of data descriptions. The description must permit fragmentation of data to modest sizes so that mainly relevant data are shipped; without a data dictionary entire files must be moved among sites if data are needed remotely.

Database systems tend to support the dictionary requirements through their *schemas*. The use of a schema distinguishes database management systems ¿From most file systems. We do find some file systems which use schemas, or at least structures closely related to schemas.

For instance, RMS for the DEC VAX VMS operating system provides, in addition to the common operations for (1) creating a file, (2) opening a file, and (3) closing a file, a number of *parameter areas* with descriptive information. This can be accessed to determine the structure of the file, as listed below:

1. The structure of the file
2. Naming of the file, including path names for distributed access
3. Record access specification
4. Disk allocation for the file
5. Dates of log creation, backup, revision, and expiration
6. Versions of the file
7. Key fields
8. Compression

9. Protection

This information can then be used to selectively extract and convert data for shipment.

Systems now in development extend such specification to permit automation of linkages between programs and files. After all, it should not be necessary for a user to specify the format of the file to be read; this information can perfectly well be obtained by the program during compilation or linking, prior to execution. If the cost of schema processing is low, the information can be interpreted during execution.

8 Operation

The data engineering process presented has led us to the definition of appropriate data structures based on data semantics. Associated with the data structures must be operations which carry out the actual computations. It is here that traditional software engineering techniques take over.

8.1 Goal directed, Event-driven, and Mixed approaches

In data processing we focus on the generation of information. The steps by which data may become information provide the path for software creation. A goal-directed process is often productive. First the desired goal attributes are defined. Then the elements they depend on are found by checking backwards. This process continues until we reach persistent base data elements, or data elements which are appropriate input. We must verify that all base data elements are available and timely.

Some data-processing applications work best if triggered from changes in the data. Such an approach is event-driven rather than goal-driven, and a forward analysis is applied. All the possible effects of a change are enumerated. Parameters are set into identity connections to avoid excessive forward branching. Derivations, triggered by multiple inputs, can establish synchronization where needed.

Failure to specify permissible time lags for derivations can lead to excessively costly systems:

G In a goal-directed approach all source data are accessed and intermediate values are computed when a result is needed. For our `Payroll` this means going to all the files on all computers which log worker activities when the paychecks are due.

E In an event-driven approach all derivations are computed when a source element changes. For us this means recomputing the `payroll` record whenever a worker checks out of the `studio`.

These extremes are not practical.

Use of identity constraints enables a mixed approach; information moves through the systems at rates determined by significance and validity. Some persistent data are derived based on changes in source values, and these are used for the computation of goal results.

We can now consider the tradeoff between computation, communication, and storage costs. Intermediate derivations permit combination of data and reduction of communication and replicated storage. Accessing reduced data locally is more economical than remote access. The redundancy of derived data also provides availability of information when direct communication is blocked. In general, the mixed approach permits a minimization of cost.

These concepts are, of course, used today implicitly in data-processing systems. Our contribution here is the formalization of the approach. The formalization in turn permits dissemination and education. It also provides a foundation for further research in this field.

8.2 Automatic Programming

Modeling creates a potential for automatic program generation. The code for a program has to satisfy three conditions

1. The program must obtain data from given data structures.

2. It has to carry out a specified data transformation.

3. It has to maintain the integrity of any data structures it modifies.

The first and the last step are considerably simplified if the data dictionary and the data model are already available. These provide many of the *invariants* which constrain the transformation of the specification into an operational program. These constraints will be complex because they are general rather than set up for one specific programming task. The reduction of these constraints to match a specification has been investigated [30]. The reusability of the model, once defined, may greatly improve the economics of automatic programming.

Note that we present here an alternative to *reusable software*. While we believe that basic computational functions should be reusable, we doubt that the benefits extend to larger programs. Adapting programs to serve a range of functions increases their complexity, and in the end reduces their reliability, maintainability, and performance to an extent where the costs exceed the benefits. The reluctance of programmers to adapt large existing programs to similar tasks is founded on the inability to communicate the models used for code and data structures [42].

9 Integration

We have presented the design process as if it were carried out by a single, cohesive organization. In practice that is neither true nor wise. The knowledge about data and processing semantics is with people spread throughout an enterprise. It is located primarily at the nodes where those applications are controlled. How we deal with the differences between these nodes is in part an open question, although some issues and directions should be listed.

It is best if each organizational unit maintains its own data dictionary and data model. The responsibility for specific elements must be strongly correlated with the authority to change and adapt them.

We now consider how strongly these systems are to be coupled. Loose coupling is achieved by specifying identity connections to span nodes. Frequently derivations must be attached, since the semantics of source attributes and desired values may differ [24].

Tight coupling, the goal of many distributed systems in the literature, requires a global model. The global model is obtained by integration of the component dictionaries and models [3, 16]. Integration is a complex process, but if formal models are made available, this task can be carried out by technical personnel without deep insight into the local idiosyncrasies.

During integration conflicts will be recognized. These can be presented to the parties for reconciliation. Irreconcilable differences can be accommodated by extending the global model. In the global model a generalization is defined, and the local models are represented by specializations. Mappings, using selective collection and derivation between each specialization and the generalization, will define the agreed-on differences.

It is important not to define compromises in integrated structures too early or too easily. It is easy to lose important semantics when the intrinsic reasons for differences among application views are not well-understood. This observation reinforces the rule that systems should not be designed by committee. The inherent function of a committee is to forge compromises. Compromises in data processing can easily lead to nonapplicability in the real world.

10 Advanced Roles for Dictionary and Model

Both the data dictionary and the data model are candidates for further expansion to capture semantics beyond the structural level. More information can be brought into the dictionary, so that most of the documentation for the programs operating on the data is contained within the dictionary.

Some *business rules* may be incorporated into the data dictionary. For instance, including in the *range* specification for a `fee` attribute the requirement that its value may not be greater than 10% of the `movie_budget` does not affect the data structures but does control decisions made in the contract department and its computer applications.

Data models can also contain business constraints that are not structural. In addition to the structural constraint which prohibits entries in the `Shipment` file for `films` if the `address` of the movie house is not listed in the `Theater` file, one can also have rules that prohibit the shipping of `films` without pre-payment from `theaters` that have a poor `credit_rating`.

Business rules which do not affect the data structures can be modified without requiring great changes to the existing structures. The authority for changes can be delegated to *owners* rather than having to remain with the *designers*.

Many business rules are local and instituted to facilitate local conventions. Such rules need not be considered in integration efforts. Business rules promulgated from a high level in the organization must, of course, be distributed and placed into the various local models. Having formal models actually simplifies their interpretation.

11 The Status of Data Engineering

The concepts we find in data engineering are derived from the formalization of experience with databases, and these in turn are derived from technologies required to deal with large data collections, shared by multiple users.

As all kinds of programs get bigger, and as modern computers can handle more data in primary storage, and as data become shared over multiple computers, we find that techniques used earlier in database systems are equally relevant outside of the pure database realm.

We have used a data-engineering approach for the design of a large system, DADAISM. That system is a collection of more than 30 modules, many of which process data according to knowledge about the data. Since DADAISM is a DBMS, the knowledge was naturally stored in a schema, and that data are obtained from external files for retrieval, or the user for updates. Within such a system much processing actually occurs, more than in the user transactions. On the other hand, the DBMS is not primarily concerned with the semantics of the data. Only the basic integrity rules, needed to manage the structure of the persistent data, need to be maintained.

12 Conclusion

In data-processing situations the use of data engineering methods is fundamentally better than a process-oriented design methodology. In such an environment the data are complex and voluminous, while the processing tasks can be handled by small and disjoint transaction programs. Using data-oriented design methods also permits the application of database management tools. These tools are still largely in a development stage.

The data model which drives this approach contains an increasing part of the knowledge needed to build effective software systems. Concepts from AI have been helpful in furthering this direction. In turn we expect that large knowledge-based programs will apply this technology in order to permit effective scaling-up of AI programs to realistic applications.

The importance of this approach is emphasized by the recent establishment of a series of International Conferences on Data Engineering, organized by a technical committee on Data Engineering of the IEEE Computer Society. They also publish a quarterly BULLETIN ON DATABASE ENGINEERING. We find here an evolutionary path from database technology into broader areas. The primary roots of data engineering lie in the field of database modeling. Database technology remains one of the more mature disciplines in what we now perceive as the broader field of data engineering. It is joined by developments in communication technology, as the infrastructure matures, so that issues at a higher level of abstraction can be addressed.

Acknowledgment

This survey was written while the first author was on a sabbatical leave from Stanford at IBM Stuttgart, the LILOG project. The first author thanks IBM Germany for the support and resources provided during his stay. The appropriateness of generalizing earlier work on database models to data structures in general was pointed out several years ago by Sheldon Finkelstein, now at Tandem Computers. Most of our support for research leading to these concepts was provided by DARPA, under contract N39-84-C211, managed by SPAWAR.

References

[1] J.R. Abrial, "Data Semantics," in *Data Base Management*, J.W. Klimbie and K.L. Koffeman (editors), North-Holland, 1974, 1-59.

[2] C.W. Bachman, "Software for Random Access Processing," in *Datamation*, April 1965, 36-41.

[3] C. Batini, M. Lenzerini, and S. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," in *ACM Computing Surveys* **18**:4, December 1986, 323-364.

[4] D.S. Batory and W. Kim, "Modeling Concepts for VLSI CAD Objects," in *ACM Transactions on Database Systems* **10**:3, September 1985, 322-346.

[5] M.L. Brodie and J. Mylopoulos (editors), *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, Springer-Verlag, 1986.

[6] S. Ceri and G. Pelagatti, *Distributed Database Design: Principles and Systems*, McGraw-Hill, 1984.

[7] S. Ceri and J. Widom, "Production Rules in Parallel and Distributed Database Environments," in *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, 1992.

[8] P.P. Chen, "The Entity-Relationship Model — Toward a Unified View of Data," in *ACM Transactions on Database Systems* **1**:1, March 1976, 9-36.

[9] P.P. Chen (editor), *Entity-Relationship Approach to System Analysis and Design*, North-Holland, 1980.

[10] D. Cheriton, "UIO: A Uniform I/O System Interface for Distributed Systems," *ACM Transactions on Computer Systems* **5**:1, 1987.

[11] J. Clifford and A. Croker, "The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans," in *Proceedings of the Third International Conference on Data Engineering*, 1987.

[12] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," in *Communications of the ACM* **13**:6, 1970, 377-387.

[13] E.F. Codd, "Further Normalization of the Data Base Relational Model," in *Data Base Systems*, Rustin (editor), Prentice-Hall, 1972, 33-64.

[14] E.F. Codd, "Extending the Database Relational Model to Capture More Meaning," in *ACM Transactions on Database Systems* **4**:4, December 1979, 397-434.

[15] B. Cox, *Object-Oriented Programming, An Evolutionary Approach*, Addison-Wesley, 1987.

[16] R. ElMasri, "On the Design, Use, and Integration of Data Models," Ph.D. Dissertation, *Technical Report* STAN-CS-80-801, Department of Computer Science, Stanford University, May 1980.

[17] R. ElMasri, J. Weeldreyer, and A. Hevner, "The Category Concept: An extension to the Entity Relationship Model," in *Data and Knowledge Engineering* **1**:1, June 1985, 75-116.

[18] S.K. Gadia, "Towards a Multi-Homogeneous Model for a Temporal Database," in *Proceedings of the Second International Conference on Data Engineering*, 1986, 390-397.

[19] T. Gruber, C. Baudin, J. Boose, and J. Weber, "Design Rationale Capture as Knowledge Acquisition: Tradeoffs in the Design on Interactive Tools," *Technical Report* 91-47, Knowledge Systems Laboratory, Stanford University, June 1991.

[20] B.T. Hailpern, "Verifying Concurrent Processes Using Temporal Logic," *Lecture Notes in Computer Science* **129**, Springer Verlag, 1982.

[21] M. Hammer and D. McLeod, "The Semantic Data Model: A Modeling Mechanism for Data Base Applications," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1978, 26-36.

[22] "Information Technology — Information Resource Dictionary Standard (IRDS) Framework," International Standard ISO/IEC 10027, International Organization for Standardization, 1990.

[23] M. Jackson, *System Development*, Prentice-Hall, 1983.

[24] W. Litwin and A. Abdellatif, "Multidatabase Interoperability," in *IEEE Computer* **19**:12, December 1986, 10-18.

[25] G.A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," in *The Psychological Review* **63**:2, March 1956, 81-97.

[26] E. Moss, *Nested Transactions, an Approach to Reliable Distributed Computing*, MIT Press, 1985.

[27] S.B. Navathe and R. Ahmed, "TSQL: A Language Interface for History Databases," in *Temporal Aspects of Information Systems*, North-Holland, 1987.

[28] "Prospectus for Data Dictionary System Standard," NBS-IR-80-2115, Application Systems Division, National Bureau of Standards, September 1980.

[29] N.S. Prywes and H.J. Gray, "Outline for a Multilist Organized System," in *Proceedings of the 1959 Annual ACM National Conference*, 1959.

[30] X. Qian, "The Deductive Synthesis of Database Transactions," Ph.D. Dissertation, *Technical Report* STAN-CS-89-1291, Department of Computer Science, Stanford University, November 1989.

[31] D. Sacca and G. Wiederhold, "Database Partitioning in a Cluster of Processors," in *ACM Transactions on Database Systems* **10**:1, March 1985, 29-56.

[32] J.W. Schmidt, "Type Concepts for Database Definition," in *Databases*, Shneiderman (editor), Academic Press, 1978, 215-244.

[33] M.J. Sergot, F. Sadri, R.A. Kowalski, F. Kriwaczek, P. Hammond, and H.T. Cory, "The British Nationality Act as a Logic Program," in *Communications of the ACM* **29**:5, May 1986, 370-386.

[34] M.E. Senko, E.B. Altman, M.M. Astrahan, and P.L. Fehder, "Data Structures and Accessing in Data Base Systems," in *IBM Systems Journal* **12**:1, 1973, 30-93.

[35] J. Fiadeiro and A. Sernadas, "Behavioral Aspects of Intelligent Knowledge-Based Information Systems," in *Temporal Aspects in Information Systems*, North-Holland, 1987.

[36] E.H. Sibley and R.W. Taylor, "Logical Structure to Physical Storage Mapping in Data Base Management Systems," in *Proceedings of the IFIP Congress*, 1971.

[37] J.M. Smith and D.C.P. Smith, "Database Abstractions: Aggregation and Generalization," in *ACM Transactions on Database Systems* **1**:1, June 1977, 105-133.

[38] R. Studer, "A Conceptual Model for Physical and Logical Time," in *Proceedings of the Sixth International Conference on Entity-Relationship Approach*, 1987.

[39] G. Wiederhold, *Database Design*, McGraw-Hill, 1987.

[40] G. Wiederhold and R. ElMasri, "Structural Model for Database Systems," *Technical Report* STAN-CS-79-722, Department of Computer Science, Stanford University, April 1979.

[41] G. Wiederhold and X. Qian, "Modeling Asynchrony: The Identity Connection," in *Proceedings of the Third International Conference on Data Engineering*, 1987, 246-250.

[42] G. Wiederhold, P. Wegner, and S. Ceri, "Toward Megaprogramming," *Communications of the ACM* **35**:12, 1992.

[43] A.H. Wilson, "Solution Patterns for Common Data Design Problems," in *Proceedings of the Third International Conference on Data Engineering*, 1987.

[44] N. Wirth, "The Programming Language PASCAL," in *Acta Automatica* **1**:4, December 1972, 241-259.