A Form-Based Approach for Database Analysis and Design

Most organizations use business forms to conduct daily operations and to communicate internally and externally with customers, suppliers, government agencies, and other entities. Traditionally, paper forms have been used and stored with obvious disadvantages of high retrieval and storage cost. With advances in computing technology, electronic forms have begun to supplant their paper counterparts. A number of form management systems have been developed [7, 11, 12, 18, 19]. Typically, these systems are closely integrated with a database management system because of the need to retain the contents of forms over time.

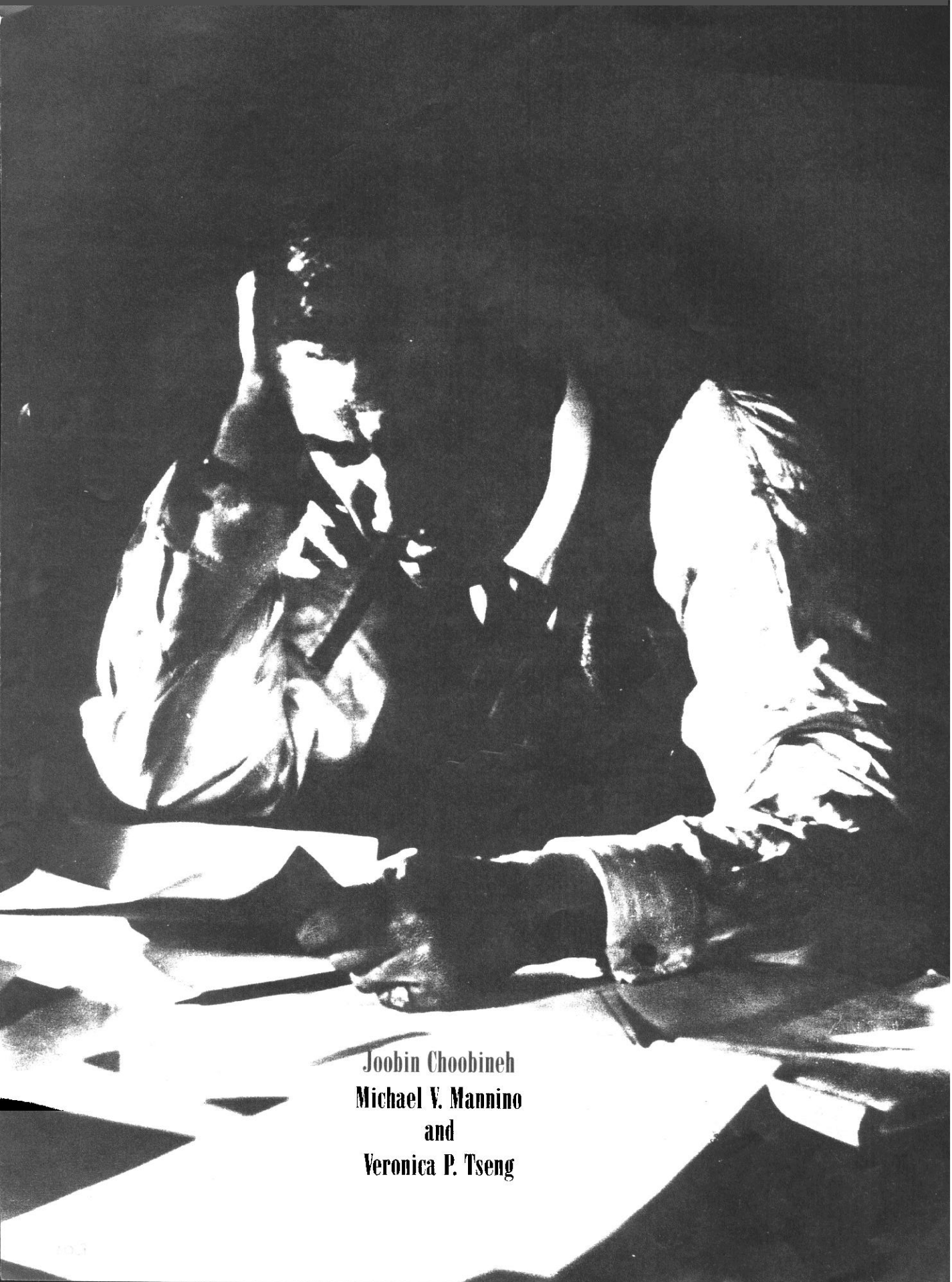
In the context of this emphasis on forms, we have conducted research on how to use forms in the database analysis and design process. This research was sparked by two observations:

(1) Because of familiarity, end users can effectively communicate many requirements through the forms they use.

(2) Usually the most widely used data are gathered or reported in a form. Thus, forms provide an important input source for database design.

Based on these observations, we have developed two systems that incorporate forms into the view definition and view integration phases of database design. The Form Definition System assists users with specifying view definitions. The novel aspect is a component that makes inferences from examples using a small collection of rules and heuristics and a purposeful dialogue. Inferences can be made about the hierarchical structure of a form as well as functional dependencies among form fields. To reduce the user's time in providing examples, the system generates some of its own examples and provides an example history facility. The Expert Database Design System uses the view definitions to incrementally build an entity relationship diagram. A collection of rules are used for grouping the fields on a form into aggregate objects (e.g., entities and relationships) and for

PHOTO: ARCHIVE PHOTOS/CAMERIQUE



Joobin Choobineh
Michael V. Mannino
and
Veronica P. Tseng

ordering a collection of forms for incremental view integration. The system applies the rules during a dialogue with a designer.

Form Model

A form is a structured collection of variables (i.e., form fields) that are appropriately formatted for data entry and display. A form type defines the structure, constraints, and presentation of the form fields. A form type may have multiple media-dependent representations known as form templates. This article only addresses character-oriented, screen templates although templates incorporating other presentation modes such as voice and bit-mapped graphics are possible. A form instance is a particular collection of values for the form fields. When a form template is filled with values, it becomes an instance of that form type. Figures 1 and 2 present form templates for the WORK ORDER and JOB ASSIGNMENT forms, respectively.

Static properties of form fields include their type, presentation, structure, origin, and constraints. The *type*, which denotes a set of values, is either a primitive type such as integer, float, or string, or a user-defined data type such as phone number. The *presentation* defines the mapping from a form field to a particular template. It includes a caption, *x-y* coordinates, and display properties such as color and font. Form fields are organized into hierarchically structured nodes. There is no limit to the number of levels or nodes in the field although most forms have a shallow (i.e., few levels) and narrow (few nodes per level) structure because of human information processing limitations. For example, the WORK ORDER form has two levels, while the JOB ASSIGNMENT form has four levels, as shown in Figure 3. The *origin* indicates the source of values for a form field. Table 1 indicates the possible origin values. For example, WORK ORDER NO on the JOB ASSIGNMENT form is form-triggered from the WORK ORDER form. The *constraints* include the designation of node keys for the form hierarchy, null value permission, default values, and value ranges, if numeric, and enumerated values if nominal.

Table 1.
Origin Values for Screen Fields

Origin Type	Symbol	Meaning
User-Triggered	U	User enters value
System-Triggered	S	System enters values without referencing any value on the form
Computation-Triggered	C	Value is computed from one or more form fields
Form-Triggered	F	Value is identically transferred from another form.
Value-Triggered	V	Value displayed because of a value of a field in current form.
Form-Value-Triggered	FV	Value displayed because of a value of a field in another form.

The dynamic properties of forms are indicated by the routing (R) of a form (F) by an agent (A) from a source station (S1) to a destination station (S2). Thus, each routing is a quadruple R(F, A, S1, S2) which is triggered by an event (E). An agent is either a human user who initiates the routing upon the perceived occurrence of the event, or a process which is triggered by some event which in turn triggers the routing. Using this model, a chain of form flow triggers can be constructed to automate office procedures. A rule definition language for form routing is described by [4].

Form Definition System

The Form Definition System supports both form and view definition. The form layout component provides a full screen editor for entering form field captions and example values. The interface component provides a Macintosh-like environment with pull-down menus, a pointing device, and bit-mapped graphics. The command component provides functions for input/output, form property definition, and explanatory feedback. Novice and expert modes are provided for defining form properties. The inference component supports the novice mode through a collection of rules and heuristics. The hierarchy subcomponent makes inferences about the grouping of form fields into nodes and the hierarchical relationship among nodes. The key/dependency subcomponent makes inferences about node keys, functional dependencies, computational dependencies, and multivalued dependencies. The view definitions (form hierarchy, node keys, and dependencies) are stored in the form abstraction base so that they are available to the Expert Database Design System.

In the novice mode, the user enters examples rather than directly stating form properties. One example is a data value in a form. Since the examples are actual or hypothetical data, they are positive examples. The inference component is a secondary source of examples. At certain points, the inference component can generate examples that can be positive or negative. An example is positive if accepted by the user, otherwise it is negative. For instance, a duplicate key value is a negative example. This approach to negative examples (similar to [9]) was chosen because the user may have difficulty directly generating negative examples.

We will explain the inference component, followed by a description of explanatory feedback that can be obtained during its use. Before presenting the inference component, basic assumptions and terms are defined.

Assumptions and Terminology

Three assumptions underly most of the rules and heuristics of the Inference Component. The first assumption constrains the manner in which a user enters examples.

Assumption 1: Users enter data in a fully nonnormalized format.

Nonnormalized data entry means that for a given parent-child relationship, the parent occurrence is entered only with the first child occurrence. This definition is recursively applied to all descendant nodes. This format provides minimum data entry.

The second and third assumptions involve the relative position of form fields within a node and the relative position of nodes.

Assumption 2: Except for the root node, fields of the same node are positioned together.

Assumption 3: Nodes on the same path are positioned adjacently from left to right where the nodes to the left are on the same or a lower hierarchical level.

In addition to the three assumptions, there are two important terms: missing value and field cardinality. Because of nonnormalized data entry, missing values can be present. A missing value is implied between a parent and its descendant nodes and unnecessary among nodes on different paths. For example, "Fiber glass" has two bins, but it is not repeated for BIN-NO 5 in Figure 2 because it is implied as a parent value. Because the hierarchical structure of Figure 2 contains only a single path, there are no unnecessary values. A missing value is contrasted with a null value be-

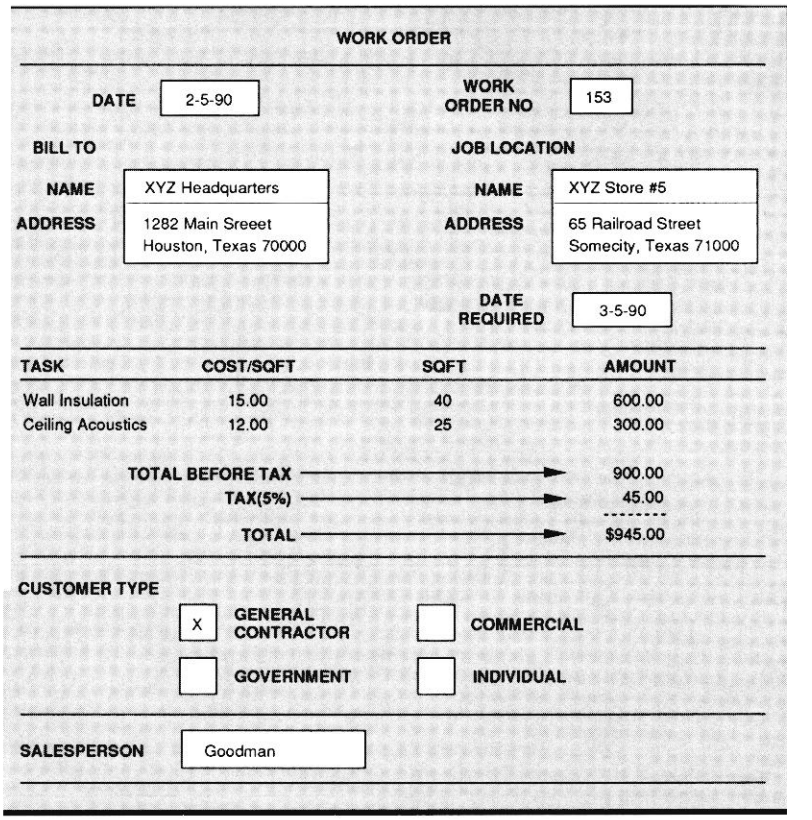


Figure 1. An instance of the Work Order Form

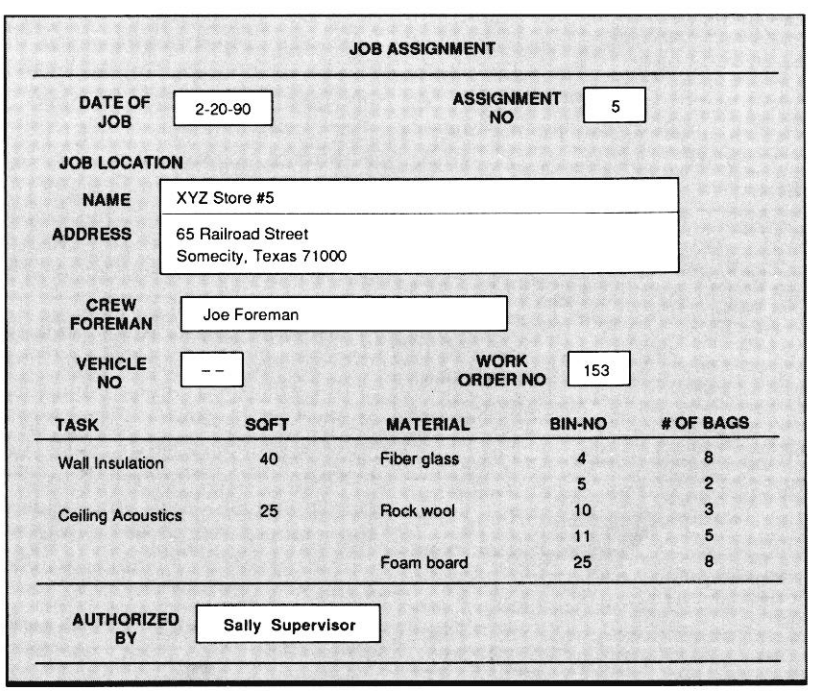


Figure 2. An instance of the Job Assignment Form

Table 2.
Summary of Hierarchy Inference Rules and Heuristics

Category	Purpose	Conditions
Field Clustering	Group Form Fields into nodes	Equivalent field cardinality Positioned adjacently 1:1 mapping among fields
Parent Child	Determine hierarchical position within a path	Smaller cardinality Missing values Node containment
Path Detection	Detect a path change	Larger cardinality No containment
Multi-Path Parent	Determine the parent of 2 adjacent nodes on different paths	Level number Missing values
Hierarchy Preference	Choose the most conservative, covering hierarchy	Level number Path number

cause a missing value is denoted by a blank and a null value by a dash. In Figure 2, the null value indicates that the VEHICLE NO is unknown.

The *field cardinality* is the number of values of a field in a set of examples including null values, but excluding missing values. For example in Figure 2, TASK, MATERIAL, and BIN-NO have cardinalities of 2, 3, and 5, respectively. After fields are clustered into nodes, we refer to the *node cardinality* which always equals the cardinality of any of its constituent fields.

Form Hierarchy Inference

The hierarchy inference component uses a collection of heuristics and rules based on the previously discussed assumptions and definitions. The heuristics suggest structures and assertions based on a given example set. They have not been proved to be correct because assertions cannot be proved from an example set. Underlying the heuristics are rules that have been proved [15]. We sometimes use the rules directly to prove that an assertion cannot hold or to make further inferences from basic assertions.

The system infers the form hierarchy in four steps:

1) Cluster form fields into nodes,
2) Identify paths and determine the hierarchical structure of each path,
3) Identify the parent nodes of multipath structures, and
4) Validate the conclusion through additional examples and generalize to the hierarchy which covers all the instances.

For each example, the system infers one hierarchy using the rules and heuristics of the first three steps. These rules and heuristics are designed to infer the simplest plausible hierarchy for an example. In the fourth step, the user provides additional examples and the system applies the rules and heuristics of the first three steps. The process terminates when the same hierarchy is inferred for two consecutive examples.

To demonstrate the inference process, we present a few examples and discuss the rules and heuristics applied. As a guide, Table 2 provides a summary of the rules and heuristics. Our intent is to provide a basic understanding of the kinds of rules rather than a detailed explanation of each. A more detailed account is given in [16, 17].

The field-clustering heuristics group fields into nodes based on their cardinality, positioning, and mapping. In Figure 2, all fields that have a cardinality of 1 are clustered together. Similarly, BIN-NO and # OF BAGS are clustered because they have identical cardinalities, are positioned adjacently, and have a 1:1 mapping among the values.

After identifying the nodes, paths of nodes and the hierarchical position of nodes within a path are determined. Two adjacent nodes are either a parent-child combination or on different paths. The parent-child rules use information about node cardinality and missing values to handle the first case. In Figure 2, the node containing TASK is selected as the parent of the node containing MATERIAL because the cardinality of TASK is smaller than the cardinality of MATERIAL, and there is a missing value for TASK where there is a value for MATERIAL.

If two adjacent nodes are not related as parent-child, they must be on different paths. The path detection rules use information about missing values and node containment to decide that two nodes are on different paths. Since the Work Order and Job Assignment forms only have a single path, the path detection rules do not apply.

To decide the parent of node $n+1$, which lies on a different path than node n, the multipath parent rule is used. This rule eliminates nodes as potential parents. Node m is eliminated as a potential parent of node $n+1$ if node m contains a value in a row where node $n+1$ has a blank. The root node is never

eliminated by this rule. If multiple parent nodes are still possible, we select the potential parent with the largest level (i.e., lowest in the tree). This is equivalent to choosing the nearest, plausible node as the parent.

For each instance provided, we apply the previously discussed rules to identify the simplest, plausible hierarchy which represents the given instance. The chosen hierarchy, however, may not be the true one because there are multiple plausible hierarchies for any form instance. Our approach is to ask for at least two instances and then to select the hierarchy that covers all the instances. The process terminates when identical hierarchies have been inferred for consecutive instances.

To help choose the covering hierarchy, we use hierarchy preference heuristics. The first heuristic gives preference to a structure with more levels and nodes but the same number of paths as an alternative structure. For example, a structure with 4 levels, 4 nodes, and 1 path is given preference over a structure with 3 levels, 3 nodes, and 1 path. A slightly different heuristic gives preference to a structure with the same number of levels but additional nodes and paths. These heuristics define the *less general than* relation meaning that if hierarchy A is *less general than* B, then every valid instance of A is also a valid instance of B. This relation is transitive so that every form instance can plausibly represent multiple hierarchies. The number of possible hierarchies is only limited by the number of fields on the form.

When we compare two instances, we choose the most general hierarchy under consideration; that is, the more general structure is the taxonomy. Applied to Figure 3b, assume that from consecutive examples we inferred (4,1,4) and (3,1,3) hierarchies where the numbers denote levels, paths, and nodes. Based on the hierarchy preference heuristics, we would choose the former structure because it has more levels and nodes but an equal number of paths.

The hierarchy preference heuristics cannot be applied to all pairs of structures. For example, there is an ambiguity between structures (4,1,4) and (2,3,4) because the former contains more levels but fewer paths than the latter. There may, however, be existing instances that plausibly represent both structures. If the preference heuristics do not apply for two instances, the system requests the user to provide another example.

Node Keys

After a hierarchy is inferred, the node and local keys are determined by identifying, ranking, and testing potential keys. A field in the root node is a potential key if it has no duplicates across form instance examples, no examples with null values, and is not a computed field. For dependent nodes, keys are formed by concatenating the key of its parent with its local key. A field is a potential local key if it has no duplicates within its immediate parent, no examples with null values, and is not a computed field.

We use several heuristics to rank potential keys. First, fields with unique values across all examples on a form are ranked the highest. Second, we further rank the first group and the remaining fields by data type: 1) integer, 2) alphanumeric, 3) alphabetic, and 4) float. Third, ties after the first two rankings are broken by a left to right preference.

The system tests the potential ranked order keys by generating additional examples to eliminate potential keys. The system generates a duplicate value in the field under test and different values for the other fields. If the user accepts the duplicate value, the potential key is eliminated from consideration. For example, in node 2 of the Work Order form, TASK, COST/SQFT, and SQFT are potential keys. AMOUNT is eliminated because it is computed. TASK is ranked more plausible than COST/SQFT because of its string data type and position. The system tests TASK by generating the last value ("Ceiling Acoustics") and different

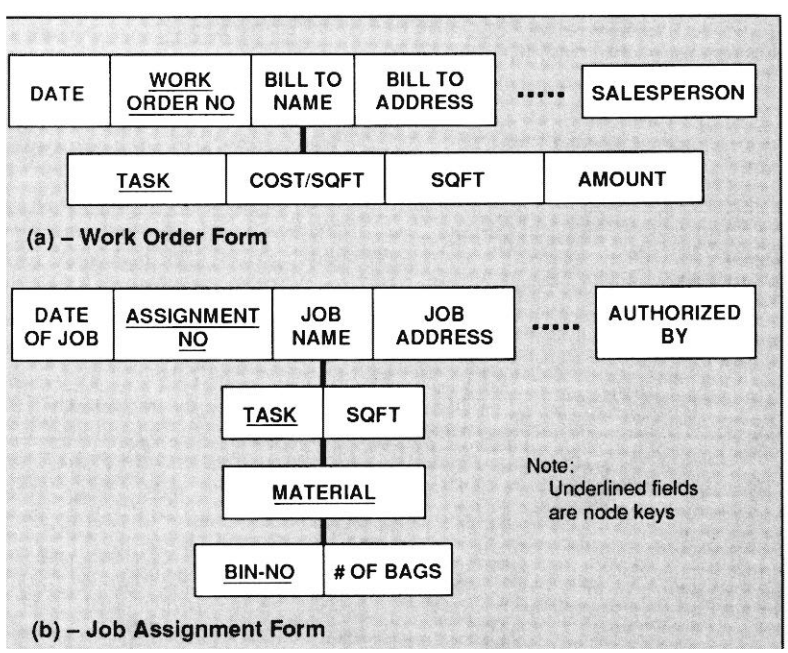


Figure 3. Hierarchical representation of form fields
a. Work Order Form
b. Job Assignment Form

values for COST/SQFT and SQFT. If the user accepts the example, TASK is eliminated as a potential key. Similarly, the system can generate new examples for COST/SQFT and SQFT.

Testing stops when all keys have been tested or when the user indicates. The user may wish to stop testing when one key is found or after some percentage of the fields are tested. The database designer can use the feedback facility to see a history of the fields tested and the keys identified.

Dependencies

Some functional dependencies and multivalued dependencies can be inferred from the hierarchical structure, the node keys, and the mathematical formulas. A functional dependency (FD) $A \rightarrow B$ means that for a given value of A, at most one B value is possible. A node key determines the other fields in its node. For the dependent nodes, the node key is always composite because of the concatenation between the local key and the node key of its parent. If there are several candidate node keys, there is a mutual dependence among them. Functional dependencies are also implied by mathematical formulas given by the user. A formula specifies a dependency between the computed field and the fields in the formula.

Multivalued dependencies always occur in pairs. The multivalued dependency (MVD) $A \twoheadrightarrow B|C$ means that each A value is associated with a collection of B and C values and the B and C value collections are independent. Further, an MVD is embedded if the relation containing A, B, and C also contains another collection of attributes which are not part of A, B, or C. Multipath structures imply MVDs (possibly embedded). If D_j and D_k are siblings with common parent P_i, the MVD $P_i.NK \twoheadrightarrow D_j.LK|D_k.LK$ holds where NK represents the node key and LK the local key. Multipath structures are not common because independence among collections of fields can be confusing.

The system generates examples to test a subset of the remaining FDs. An example eliminates the FD, $A \rightarrow B$ if two rows agree on the same A value but differ on the B value. The system generates examples by holding the determinant constant and creating a new value for the determined field. If the user accepts the example, the potential FD is eliminated. Users can also augment the initial examples with new examples. In addition, the user can cut and paste previous examples using the *example history* function.

Because the number of potential, remaining FDs is large, the Form Definition System only generates examples to test FDs with a single field determinant (see [8] for a recent approach to test for a cover of FDs). Even with this restriction, the number of potential, remaining FDs in a node is the number of permutations of N fields taken two at a time (P_2^N). If this level of testing proves to be too much of a burden, the user can stop at any point. It may be more practical not to try for exhaustive requirements collection directly from the user. The designer can later examine the state of the testing process using the feedback facility and take appropriate action.

As an example, consider testing functional dependencies among the fields of node 2 of the Work Order Form (Figure 3a). Assume that TASK is the only local key and AMOUNT is computed. Thus, there are six remaining FDs (i.e., 3!/(3-2)!) with a single field determinant. The given example set in Figure 1 does not eliminate any of the remaining FDs so the system generates additional examples. For each possible determinant (TASK, COST/SQFT, SQFT), the system generates examples with the last determinant value paired with a different determinee value. For example, to test the dependency SQFT → COST/SQFT, a new row is generated with a "do not care" value for TASK, 25 for SQFT, and a new value for COST/SQFT.

Informative Feedback

Informative feedback is provided in two categories: the background knowledge for making inferences and the inferences made on a given form. For the background knowledge, the system provides the definitions of the important terms such as field, field cardinality, node, root node, dependent node, numbering scheme for nodes, and so on. Visual displays are used for explaining concepts related to the hierarchical structure of a form.

The system offers explanatory feedback for all the form properties it infers. The feedback is organized by major (hierarchical structure, keys, dependencies) and minor categories (e.g., node clustering and parent-child relationships). Within each category, there are three levels of detail. The highest level of explanation provides a summary of the results. The next level of detail presents the rules and heuristics applied during the inferencing process. The lowest level lists a complete trace of the inferencing steps. The following is the lowest level of explanation generated by the Form Definition System for the field clustering heuristics applied to the Work Order instance in Figure 1.

****Explain Hierarchy****
Form Name: WORK ORDER
Instance number 1:
A. Field Clustering.
Fields with the cardinality of one are grouped into a root node.
Fields that have the same cardinality, are positioned adjacently, and have a 1:1 mapping in every row are grouped into a dependent node.
The following fields have a cardinality of 1, therefore, they are grouped into the root node.

DATE
WORK ORDER NO
BILLTO NAME
BILLTO ADDRESS
JOB NAME

JOB ADDRESS
DATE REQUIRED
TOTAL BEFORE TAX
TAX
TOTAL
CUSTOMER TYPE
SALESPERSON

The following fields have cardinality of 2, are adjacent to each other, and have a 1:1 mapping on every row. Therefore, they are grouped into the same dependent node.

TASK
COST/SQFT
SQFT
AMOUNT

Expert Database Design System (EDDS)

The Expert Database Design System (EDDS) produces an Entity-Relationship Diagram (ERD) based on the analysis of the forms contained in the Form Abstraction Base [2]. EDDS incrementally builds a schema diagram by analyzing one form at a time. A collection of rules is used to determine the order in which the forms are analyzed and to identify the entities, attributes, and relationships that represent the forms. The system applies the rules during a dialogue with a designer. The rules help a designer derive a consistent schema diagram that represents the forms. It should be noted, however, that not all of what may be included on an ERD of an enterprise may appear on the set of its forms. The ERD (which is derived from the set of the forms) may have to be modified or augmented to correctly and completely represent the enterprise's data schema. In [10] we defined a restricted natural language to support the analysis of forms in conjunction with EDDS.

Similar to other works [1, 6, 13], our expert system is designed to aid in the documentation, reorganization, and consistency and completeness checking of the design. Furthermore, due to its consultative nature, it aids a designer in making decisions and in highlighting the possible design alternatives. The designer may confirm or disconfirm any of the software's suggestions as the schema is being developed in a session.

The Architecture of EDDS

The EDDS contains a knowledge base and three databases. The Data Design Knowledge Base (DDKB) contains general data design rules as well as rules for mapping from the forms to ERDs. The Form Abstraction Base (FAB) contains the form definitions as discussed previously. The Design Database (DDB) contains the evolving schema diagram (i.e., an ERD). The Design Status Base (DSB) records the current status and past design decisions. The three databases (FAB, DSB, and DDB) enable continuity of work over time and between different sessions of the same design.

The inference engine is a data-driven rule interpreter. It matches the facts in the three databases (FAB, DSB, and DDB) to the antecedents of the rules in DDKB. When in doubt about a design decision, the inference engine asks for confirmation from the human designer. Depending on the contents of the databases and the designer's response to inquiries, the inference engine decides which rule to fire next. A rule is fired whenever an assertion is matched to the antecedent of the rule. As a design session progresses, the ERD for the collection of forms is gradually evolved.

The inference engine uses six distinct groups of rules corresponding to the six phases of database design supported by the EDDS. The *form selection phase* determines the next form to analyze. The *entity identification phase* determines form fields that represent entities. The *attribute attachment phase* adds attributes to the previously identified entities, while the *relationship identification phase* connects the previously identified entities with relationships. The *cardinality identification phase* makes decisions on the minimum/maximum cardinalities of an entity in a relationship. The *consistency phase* ensures the consistency of the evolving schema diagram. Each phase corresponds to one group of rules. Some rules cross the boundaries between phases in that their consequents are not limited to a single phase. The collection of the six groups of rules constitute the Data Design Knowledge Base.

Across all six phases, we can divide the rules into two groups. One group is a collection of mappings and decisions which are made by the system without affirmation from our consultation with the designer. The second group of rules prompt the consequent of the rule as a suggestion to the designer. The designer then either rejects or accepts the system's suggestion.

The system operates on one form at a time using the knowledge about previously analyzed forms which are contained in DDB. Initially, a form to be analyzed is chosen and an entity relationship diagram that represents the form is derived according to the rules of the next five phases. Another form is then chosen for analysis. The previously designed schema is now augmented with the result of the analysis of the current form. This process continues until no more form fields remain and all the requirements of the last three groups of rules are satisfied.

Figure 4 is the ERD output from EDDS for the integrated Work Order and Job Assignment Forms. Entities and relationships are shown by rectangles and diamonds, respectively. The minimum/maximum cardinalities are shown by separating the two by a colon and enclosing them in parentheses. As an example of the interpretation of min/max cardinalities, consider the relationship between SALESPERSON and WORKORDER. The (0:m) for SALESPERSON means that a salesperson can issue zero or more work orders. The (1:1) for WORKORDER means that a work order must be associated to exactly one salesperson.

Form Analysis

We now describe the rules underlying each of the design phases. Most of the rules are mappings from the

form model to an instance of an ERD. We will present some representative rules in the following subsections. To make a more readable presentation, pseudocode in the form of the IF...THEN... rules will be used. (The actual code was written in Pascal. See [3] for more details.) Using Figure 4, examples of application of some of the rules will be given. For a complete list of the rules and their discussion, see [3].

Form Selection. The form selection phase determines the next form to analyze using the origin types (see Table 1). The form selection decisions are made by EDDS without consultation with the designer. The system considers two cases. The first case is when no forms have been analyzed or when no forms are related to the collection of previously analyzed forms. In this case, the form with zero destination fields[1] is chosen. If more than one form satisfies this criterion, the form with the largest number of V (i.e., Value) fields is chosen because the analysis of V fields is straightforward (see the subsection on Attribute Attachment).

The second case considers the remaining forms that have fields originating from the previously analyzed forms. The rationale is to maximize the number of destination fields whose source is on previously analyzed forms. The next form to analyze is the one most closely related to the previously analyzed forms. The strength of the relationship between form B and form A is measured by the number of destination fields in B that have their sources in A. The number of V fields is used, as described in the previous paragraph, to resolve cases where the strength is equal.

As an example, suppose that WORK ORDER does not have any destination fields. JOB ASSIGNMENT has the following destination fields: WORK-ORDER-NO, JOB-LOCATION, TASK, and SQFT where the origin type of all these fields is F and their sources are from the WORK ORDER form. Thus, WORK ORDER is chosen as the first form to analyze. Suppose that there is another form called DAILY CREW ASSIGNMENT containing information about the crew members for each job. Further assume that the WORK-ORDER-NO, TASK, and SQFT are three fields on this form whose sources are in the WORK ORDER form. According to the form selection rule, the JOB ASSIGNMENT form will be the next form to analyze since it has one more destination field whose source is on the previously analyzed forms than the DAILY CREW ASSIGNMENT form.

The following two rules are pseudocode for the actual rules. In these rules, PF is the list of previously analyzed forms, RF is the list of remaining forms, QF is the list of qualifying forms, and F is the final form selected. The predicate DEST-FIELDS-IN-RF returns true if at least one form in its second argument contains a destination field whose source is in its first argument. The predicate SMALLEST-NUMBER-F-FV-FIELDS returns a list of forms in its second argument that have the smallest number of F and FV fields. The predicate LARGEST-NUMBER-V-FIELDS returns a form name in its second argument from among the forms in its first argument, which has the largest number of V fields. If more than one form qualifies, one is arbitrarily chosen. The result of both rules is to make F the current form and to delete F from the remaining forms list.

```
IF (NOT (DEST_FIELDS_IN_RF
  (PF, RF))) —no related forms
  AND
  SMALLEST_NUMBER_F_FV_
  FIELDS (RF, QF) AND
  LARGEST_NUMBER_V_
  FIELDS (QF, F)
THEN
  ASSERT (IS_FORM_TO_
  ANALYZE (F))
  DELETE (F, RF)
IF DEST_FIELDS_IN_RF (PF, RF)
  —some related forms
  AND
  LARGEST_NUMBER_F_FV_
  FIELDS (PF, RF, QF) AND
  LARGEST_NUMBER_V_
  FIELDS (QF, F)
THEN
  ASSERT (IS_FORM_TO_
  ANALYZE (F))
  DELETE (F, RF)
```

Entity Identification. To identify possible entities on a form, we use heuristic rules that are based on the local keys, dependencies, origin, name, and grouping of form fields. These rules suggest that a form field may represent an entity. The designer is asked to confirm the suggestion. Informal examples of these rules follow:

- Any form field designated as a determinant in a functional dependency represents an entity (e.g., TASK-NAME representing entity TASK).
- Form field(s) on the left or right-hand sides of the MVD may represent an entity.
- Any form field designated as a local key may represent an entity.
- Any form field matching a common candidate key suffix such as NAME, NO, NUMBER, or # may represent an entity (e.g., WORK ORDER NO representing the entity WORK-ORDER).
- A group form field may represent an entity. Examples of group fields are BILL TO and JOB LOCATION, which represent CUSTOMER and JOB-LOCATION entities. The subfields of the group are attributes of the identified entity.
- Any form field that is the source of another form field, whether of this form or another, may represent an entity.

As an example, consider the group/subfield rule (fifth in the preceding list):

```
IF INLIST (FF, RFF) AND
  SUB_FIELD (FF) AND
  (ORIGIN (FF, 'U') OR
```

[1]Form fields whose origin types are F (form triggered) or FV (form-value triggered) are called destination fields.

```
    (ORIGIN (FF, 'S') OR
    (ORIGIN (FF, 'C'))
  THEN
    MAY_REPRESENT_ENTITY
    (FF, E)
```

FF, RFF, and E represent a form field, the list of remaining form fields, and a to-be-decided entity name, respectively. The result of firing the rule is a prompt to the designer as to whether FF represents an entity. A positive response is followed by another prompt for the name of this entity. The rule concludes by adding the new entity E to the list of previous entities, attaching FF as an attribute of E, designating FF as the candidate key for E, and deleting FF from the list of the remaining form fields. An associated rule attaches other subfields of this group field to E (see the subsection, Attribute Attachment).

Attribute Attachment. In this phase, attributes of entities and relationships are identified. The most direct rules are those that use functional dependencies. For example, there is a rule that makes both V and FV fields attributes of the same entity as their source field because of the functional dependencies of the V and FV fields on their source fields. In Figure 4, COST/SQFT is attached to the TASK entity by this rule. The pseudocode for this rule follows:

```
IF INLIST (FF1, RFF) AND
  ATTRIBUTE_OF (FF2, E)
  AND
  SOURCE (FF2, FF1)
  AND
  (ORIGIN (FF1, 'V')
  OR
  ORIGIN (FF1, 'FV'))
THEN
  ATTACH (FF1, E)
```

As another example, consider a rule based on the physical proximity of a form field to the previously identified entities. This rule is based on a heuristic that suggests related form fields are often clustered together. The closer a form field is to the collection of other

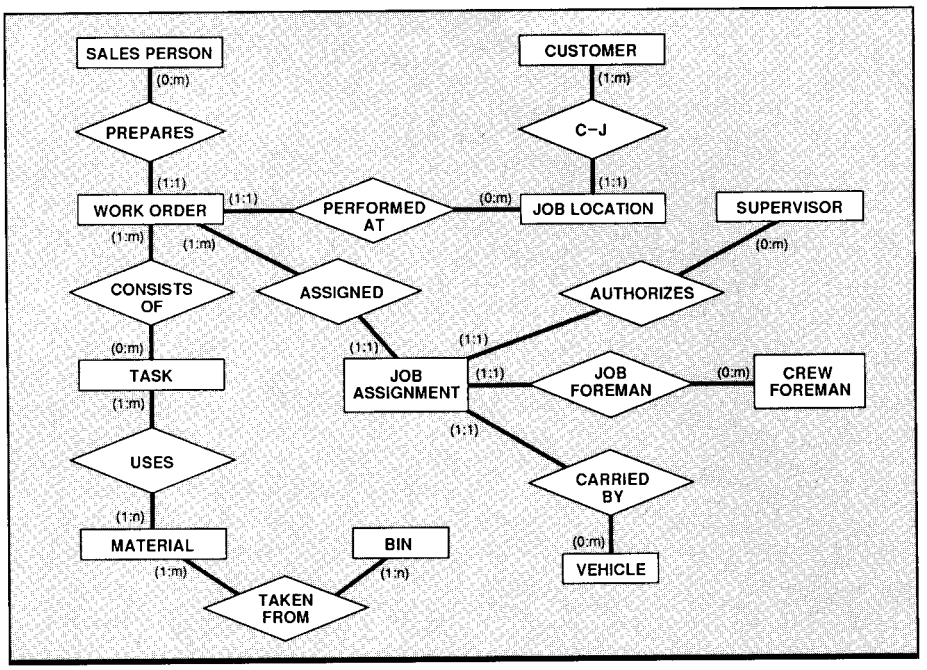


Figure 4a. The ERD for the Integrated Work Order and Job Assignment Forms

ENTITIES	
SALESPERSON	NAME
CUSTOMER	NAME, ADDRESS, TYPE
WORKORDER	DATE, WORK-ORDER-NO, DATE-REQUIRED, TOT-BEF-TAX, TAX-RATE, TAX, TOTAL
JOBLOCATION	NAME, ADDRESS
SUPERVISOR	NAME
JOBASSIGNMENT	ASSIGNMENT-NO, DATE-OF-JOB
TASK	TASK-NAME, COST/SQFT
MATERIAL	NAME
VEHICLE	VEHICLE-NO
BIN	BIN-NO
CREWFOREMAN	NAME
RELATIONSHIPS	
CONSISTSOF	SQFT, AMOUNT
TAKENFROM	#OFBAGS

Figure 4b. Attributes of entities and relationships

fields on an entity or a relationship, the more likely it is that this field is another attribute of that entity or relationship. The proximity factors are computed from left to right and top to bottom on a form. The proximities of the remaining form fields to each of the previously identified entities and relationships are recomputed each time a form field is removed from the list of the remaining form fields.

To illustrate the proximity heuristic, suppose the form field ASSIGNMENT NO of Figure 2 has identified the entity JOB ASSIGNMENT of Figure 4a. The proximity factors for the DATE OF JOB and JOB LOCATION NAME to the JOB ASSIGNMENT entity are both 1. The proximity factors for JOB LOCATION ADDRESS and CREW FOREMAN are, respectively, 2 and 3. Now, assume that the JOB LOCATION is identified as an entity with the two attributes JOB LOCATION NAME and JOB LOCATION ADDRESS. These two attributes, therefore, will no longer be in the list of the remaining form fields. The proximity factors for the two entities JOB ASSIGNMENT and JOB LOCATION are now recomputed for all the remaining form fields. This factor is equal to 1 for both pairs of ⟨DATE OF JOB, JOB ASSIGNMENT⟩ and ⟨CREW FOREMAN, JOB LOCATION⟩. Looking at the proximity factors, the designer decides which form field, if any, should be an attribute of JOB ASSIGNMENT or JOB LOCATION. In this case, as is shown in Figure 4b, the designer decided that the DATE OF JOB is an attribute of JOB ASSIGNMENT.

For each remaining form field, which cannot be attached by any of the rules, the designer can define a new entity or relationship, attach the form field to an existing entity or relationship, or leave the form field on the list of remaining form fields. The first two choices will always trigger the chain of rules again.

Relationship and Cardinality Identification. Relationships and cardinalities are identified from the origin of form fields, hierarchical structure of the forms, and both functional and multivalued dependencies. A relationship is established between two entities where a form field in one functionally determines a form field in the other. The maximum cardinalities will be 1 and *m* for the former and the latter entities, respectively. The PREPARES relationship between WORK ORDER and SALES PERSON is an example of this rule since WORK ORDER NO determines NAME of SALES PERSON. The pseudocode for this rule is as follows:

```
IF ATTRIBUTE_OF (FF1, E1)
   AND
   ATTRIBUTE_OF (FF2, E2)
   AND
   SOURCE (FF2, FF1)
   AND
   (ORIGIN (FF2, 'V')
   OR
   ORIGIN (FF2, 'FV'))
THEN
   ESTABLISH_RELATIONSHIP
   (E1, E2, R, ROL1, ROL2)
   ASSERT (MAX_CARD
   (E1, R, '1'))
   ASSERT (MAX_CARD
   (E2, R, 'm'))
```

ROL1 and ROL2 are, respectively, the role names of E1 and E2 in relationship R. The designer is prompted to provide the names for these roles.

In a multivalued dependency, separate relationships are established between each of the determinees and the determinant. The maximum cardinality of the determinant is *m* in both relationships.

When no relationships, either direct or indirect, have been identified between two adjacent levels *i* and *i*+1, the designer is prompted to identify one. In the identified relationship, the max cardinality of the entity at level *i* will be *m*. CONSISTS OF is an example of a direct relationship between two levels of a form. There is an indirect relationship between JOB ASSIGNMENT and TASK through ASSIGNED and CONSISTS OF relationships.

Entities on two different forms are related to each other by using the information on local keys and the origin types of form fields. If a form field is a root key, and it is also the origin of an F type field on a second form, then there could be a relationship between the entity represented by this root key to an entity in the second form which contains its root key. The ASSIGNED relationship between WORK ORDER and JOB ASSIGNMENT is identified through this rule.

While identifying the relationships between entities, the designer is asked to specify the role name for each entity in the relationship. The role names are used to identify reflexive relationships (i.e., a relationship between the same entity) and multiple relationships between the same two entities.

To preserve the functional dependencies implied by node keys which are on the same level of a form, all other form fields in the same node must either be an attribute of the same entity as the node key, or be an attribute of another entity which is functionally related (directly or indirectly) to the entity of the node key. Entity E1 is directly functionally related to Entity E2 in relationship R1 if the maximum cardinality of E1 with respect to E2 is one. Entity E3 is indirectly functionally related to E1 if there exists a relationship R2 between E2 and E3 and the maximum cardinality of E2 with respect to E3 is one.

Finally, there are rules for assigning cardinality of 1.

- If a field of an entity is a node key, then the minimum cardinality of this entity is '1' in a relationship to an entity that has a form field on the same node and for which no nulls are allowed.
- An identification-dependent entity (an entity that is existent-dependent and not self-identified)

is assigned a 1 for both its minimum and maximum cardinalities.

Integrity Constraints. Two types of integrity rules are applied to check the consistency and completeness of the evolving ER diagram. The form-mapping constraints check the mapping from the form model to the evolving schema diagram. The first form-mapping constraint ensures there must be at least one relationship, direct or indirect, connecting entities in adjacent levels of a form. We give the pseudocode for this rule. Here, the predicate LEVEL(FF, L) designates the level (L) of the form field FF on its form.

```
IF ATTRIBUTE_OF (FF1, E1)
  AND
  ATTRIBUTE_OF (FF2, E2)
  AND
  IN_SAME_FORM (FF1, FF2)
  AND
  LEVEL (FF1, L1)
  AND
  LEVEL (FF2, L2)
  AND
  EQUAL (+(L1, 1), L2)
  AND
  NOT (RELATIONSHIP_
  BETWEEN (E1, E2, R))
THEN
  ESTABLISH_RELATIONSHIP
  (E1, E2, R, ROL1, ROL2)
```

The second form-mapping constraint ensures that the implied functional and multivalued mappings between form fields are maintained in the evolving schema diagram. The third form-mapping constraint ensures that all form fields, except those with an origin of F, are represented on the diagram.

The second collection of integrity rules enforce constraints on the ERD. Some of the integrity rules such as the uniqueness of entity and relationship names and the requirement of unique role names for each entity in two or more relationships are enforced as the diagram is constructed. Other rules are enforced after the designer indicates that a form analysis is complete. Examples of these rules are:

1) Each entity must have at least one candidate key except for identification-dependent entities,

2) All entities involved in a relationship must have an associated minimum-maximum cardinality, and

3) One candidate key of each entity must be designated as the primary key.

Implementation and Usage Status

Prototype versions of the Form Definition System and the Expert Database Design System have been implemented. These systems were originally implemented in Pascal on a Vax 11/780 [3]. In the original design, the Form Definition System did not provide an inference component for the novice user. In later research [15], the Form Definition System was redesigned and reimplemented in Lightspeed Pascal [14] on a Macintosh SE. The standard features of the Macintosh environment, such as pull-down menus, dialogue boxes, and windows, were utilized in the user interface.

The Form Definition System was tested in a preliminary study using subjects from the Computing Services Division of the University of Massachusetts at Boston [15, 17]. The purpose of the study was to collect evidence about the types of mistakes made, the ability of novice users to provide examples of requirements, and the completeness of collected requirements. We observed that the system was useful in educating users and collecting requirements, but that some dependencies were missed by the novice users. We concluded that the Form Definition System is most useful in providing a common vocabulary and goals among end users and data processing professionals, rather than in providing exhaustive requirements collection by end users.

Conclusion

We described an approach to systematically use electronic forms in the database requirements and design processes. The foundation of our study was a simple form model that includes hierarchically structured forms with an event-driven routing. The Form Definition System provides an inference component to assist an end user with view definitions for their forms where a view definition consists of the hierarchical structure and functional dependencies among form fields. The inference component uses a collection of rules and heuristics along with a purposeful dialogue. An explanation facility provides feedback at several levels of detail. The Expert Database Design System assists a designer in the view integration process. The system provides rules for incrementally integrating the form views and heuristics for mapping the form fields into entity types and relationships.

We believe that forms provide an important input to the database design process that should be formalized into existing database design methodologies. The form-based approach is especially relevant when forms are important in the database and end users are accustomed to form-based work. For completeness, the approach described here should be combined into a database design methodology that permits input from other sources in addition to forms such as natural language descriptions.

Acknowledgments

The work of Joobin Choobineh was partially supported by the Center for Management of Information Systems, Texas A&M University.

References

1. Bouzeghoub, M., Gardarin, G., and Metais, E. The design of an expert system for database design. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Stockholm, Sweden. (Aug. 1985).

2. Chen, P. The entity-relationship model: Toward a unified view of data. *ACM Trans. Database Syst. 1*, 1 (Mar. 1976).
3. Choobineh, J. Form-driven conceptual data modeling. Ph.D. dissertation, Dept. of Management Information Systems, University of Arizona, 1985.
4. Choobineh, J. FORMFLEX: A user interface tool for forms definition and management. In *Human Factors in Manage. Inf. Syst.*, J. Carey, Ed., Ablex, Norwood, NJ 1988, pp. 117–133.
5. Choobineh, J., Mannino, M., Nunamaker, J., and Konsynski, B. An expert database design system based on analysis of forms. *IEEE Trans. Softw. Eng. 14*, 2 (Feb. 1988), 242–253.
6. Eick, C. and Lockeman, P. Acquisition of terminological knowledge using database design techniques. In *Proceedings of ACM SIGMOD Conference* (Austin, Tex., May 1985), pp. 84–94.
7. Embley, D. NFQL: The natural forms query language. *ACM Trans. Database Syst. 14*, 2 (June 1989), 168–211.
8. Flach, P. Inductive characterization of database relations. *Methodolog. Intell. Syst. 5*, Z.W. Ras, M. Zemankova, and M. Emrich, Eds., North-Holland, Amsterdam, 1990, pp. 371–378.
9. Mannila, H. and Raiha, K. Design by example: An application of Armstrong relations. *J. Comput. Syst. Sci. 33*, 2 (1986), 126–141.
10. Mannino, M., Choobineh, J., and Hwang, J. Acquisition and use of contextual knowledge in a form-driven database design methodology. In *Proceedings of the Fifth International Conference on Entity-Relationship Approach* (Dijon, France, Nov. 1986).
11. Shu, N. Formal: A forms-oriented, visual-directed application development system. *IEEE Comput.* (Aug. 1985), 38–49.
12. Shu, N., Lum, V., Wong, H., and Chang, C. Specification of forms processing and business procedures for office automation. *IEEE Trans. Softw. Eng. SE-8*, 5 (Sept. 1982), 499–511.
13. Storey, V. and Goldstein, R. A methodology for creating user views in database design. *ACM Trans. Database Syst. 13*, 3 (Sept. 1988), 305–388.
14. Think Technologies. *Lightspeed Pascal: User's Guide and Reference Manual*, Version 1, First ed., 1986, Lexington, Mass.
15. Tseng, V. Inferring database requirements from examples in forms. Ph.D. dissertation, Dept. of Management Science and Information Systems, The Univ. of Texas at Austin, May 1988.
16. Tseng, V. and Mannino, M. Inferring database requirements from examples in forms. In *Proceedings of the Seventh International Conference on Entity-Relationship Approach* (Rome, Italy, Nov. 1988), pp. 251–265.
17. Tseng, V. and Mannino, M. A method for database requirements definition. *J. Manage. Inf. Syst.* (Winter 1989).
18. Tsichritzis, D. Form Management. *Commun. ACM 25*, 7 (July 1982), 453–478.
19. Yao, B., Hevner, A., Zhongzhi, S., and Luo, D. FORMANAGER: An office forms management system. *ACM Trans. Off. Inf. Syst. 2*, 3 (July 1984), 235–262.

CR Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies*; H.2.1 [**Database Management**]: Logical Design—*data models, schema and subschema*; H.2.3 [**Database Management**]: Languages—*data description languages (DDL)*

General Terms: Design, Languages

Additional Key Words and Phrases: Form processing, view definition, view integration

About the Authors:

JOOBIN CHOOBINEH is an associate professor of management information systems at Texas A&M University. His research interests include conceptual data modeling, integration of data and mathematical models, application of artificial intelligence techniques to database design processes, and expert databases systems. **Author's Present Address:** Department of Business Analysis and Research, College of Business Administration and Graduate School of Business, Texas A&M University, College Station, TX 77843–4217. email: j0c1099@tamsigma.bitnet

MICHAEL V. MANNINO is an assistant professor in the Department of Management Science, University of Washington, Seattle. His research interests include database management, software engineering and knowledge representation. **Author's Present Address:** Department of Management Science, University of Washington, Seattle, WA 98195. Internet: zmann@u.washington.edu

VERONICA P. TSENG is a staff programmer at IBM Rochester, Minn. Her research interests include object-oriented programming and data modeling. **Author's Present Address:** CIM Systems, IBM Application Business Systems, Rochester, MN 55901. email: tseng@rchvmp.iinus1.ibm.com