Extracting Domain Semantics for Knowledge Discovery in Relational Databases

Roger H. L. Chiang[1], Terence M. Barron[2], Veda C. Storey[2]

[1] Information Management Research Centre, School of Accountancy and Business
Nanyang Technological University, Nanyang Avenue, Singapore 2263
[2] William E. Simon Graduate School of Business Administration
University of Rochester, Rochester, NY 14627, USA

Abstract

Our research investigates how domain semantics are discovered from a relational database at a high level of automation. The discoveries are then represented as the extended Entity-Relationship schema [2] (i.e., conceptual schema) and integrity constraints of the database, which can support and drive the subsequent knowledge discovery in this database.

This paper presents a discovery process which obtains an extended Entity-Relationship schema from a relational database by analyzing not only data instances but also the executable data schema of the database. In addition, our research appears to be the first ones to address the problem of erroneous data for discovering inclusion dependencies. An interactive knowledge-based system, the Knowledge Extraction System (KES), has been developed to perform the discovery process. KES demonstrates how knowledge-based systems technology can be applied to support the work of knowledge discovery in a database. It also illustrates that the discovery process can be implemented at a high level of automation.

1 Introduction

In discovering knowledge from an existing database, a major difficulty is that often the meaning of the data has been lost during various evolutions and modifications of the database over many years by many persons. No one may know exactly what the data and the relationships between the data really are. However, it is essential and important for any knowledge discovery process to have such an understanding in order to discover knowledge from databases. This understanding can only be achieved by raising the level of abstraction above that of the database itself and representing it as a schema in a conceptual model. Our research thus has two objectives: *1) to consider how domain semantics are discovered for existing relational databases, and 2) to develop a discovery process that obtains an extended Entity-Relationship (EER) schema that corresponds to the* **possible** *(most likely) design specifications of an existing relational database by analyzing not only the extension (data instances) but also the intension (data schema) of the database.* The term "domain semantics" refers to information about the application domain that should be captured during the requirement specification phase of database design [2].

During the design and maintenance of a database, some domain semantics may not be captured, or they may be captured but removed due to the representation limitations and implementation considerations of the database system [9]. It is often difficult to obtain a good conceptual understanding of a legacy database, especially when there is a lack of documentation and one can only refer to information provided by the target database management system (DBMS). However, legacy databases typically contain a large volume of data instances for knowledge discovery. It is necessary for any process applied to knowledge discovery in databases to understand the relationships between the data. In addition, domain semantics can also help the discovered knowledge be more meaningful to the end user [12]. Therefore, a process to obtain the domain semantics of databases at a high level of automation is obviously necessary to drive and support knowledge discovery in databases.

Several researchers have provided some means for inferring functional and inclusion dependencies from data instances of a relational database, e.g., [1, 10, 11]. However, these dependencies mainly represent the properties of the physical implementation of the database. In addition, due to the various semantics degradations during the design and maintenance of a database, any process for discovering domain semantics must have other knowledge than the functional and inclusion dependencies. For example, knowledge to make the appropriate reverse schema transformation from the relational data schema to a conceptual one is needed. Database reverse engineering provides solutions for these problems [5, 9, 13]. It produces a sufficient understanding of an existing database and its application domain by recovering the domain semantics of an existing database and representing them as a conceptual schema that corresponds to the most likely design specifications of the database. Thus, in this paper, we discuss how the results of reverse engineering research can be applied to the discovery of knowledge in a database.

This paper is divided into four sections. Section 2 introduces the discovery process together with its steps and rules. The architecture of the Knowledge Extraction System is presented in Section 3. We offer some concluding remarks in Section 4.

2 The Discovery Process

The discovery process deals with rules for discovery, the executable schema and data instances in the existing database, and the user if necessary. The process is divided into six sequential steps as shown in Figure 1. Each step is discussed briefly below. Our paper presents several rules used in the generation and identification steps of the process. Full details of the process and the rules used can be found in Chiang [5].

Sources for Discovering Domain Semantics. Sources of information for discovering domain semantics are classified into three categories according to how they can be obtained. Furthermore, in order to minimize human involvement, the order of these sources is also the sequence in which the discovery process obtains the required information.

1. **Data dictionary of the target DBMS:** Design specifications, such as the executable schema, can be obtained directly from the target DBMS's data dictionary.
2. **Analysis of the executable schema and data instances:** Information can be obtained by analysis of data instances, such as functional and inclusion dependencies, key attributes, and integrity constraints.
3. **The users:** User involvement is necessary whenever required information is neither stored in the target DBMS nor obtainable by analysis of the executable schema and data instances. The user can be a database administrator, a database designer or system analyst, or even the end-user.

2.1 Initialization of Data Schemas

The information needed about the executable schema includes: relation names, attribute names, attribute domains (data types), and primary keys. In general, the relation and attribute names can be obtained directly by querying the data dictionary. However, many current DBMSs do not store information about primary keys, e.g., DB2. Thus, the identification rule uses attributes' properties (e.g., non-null, uniquely indexed) and names (e.g., names with a prefix or suffix, such as `SSN`, `NO`,

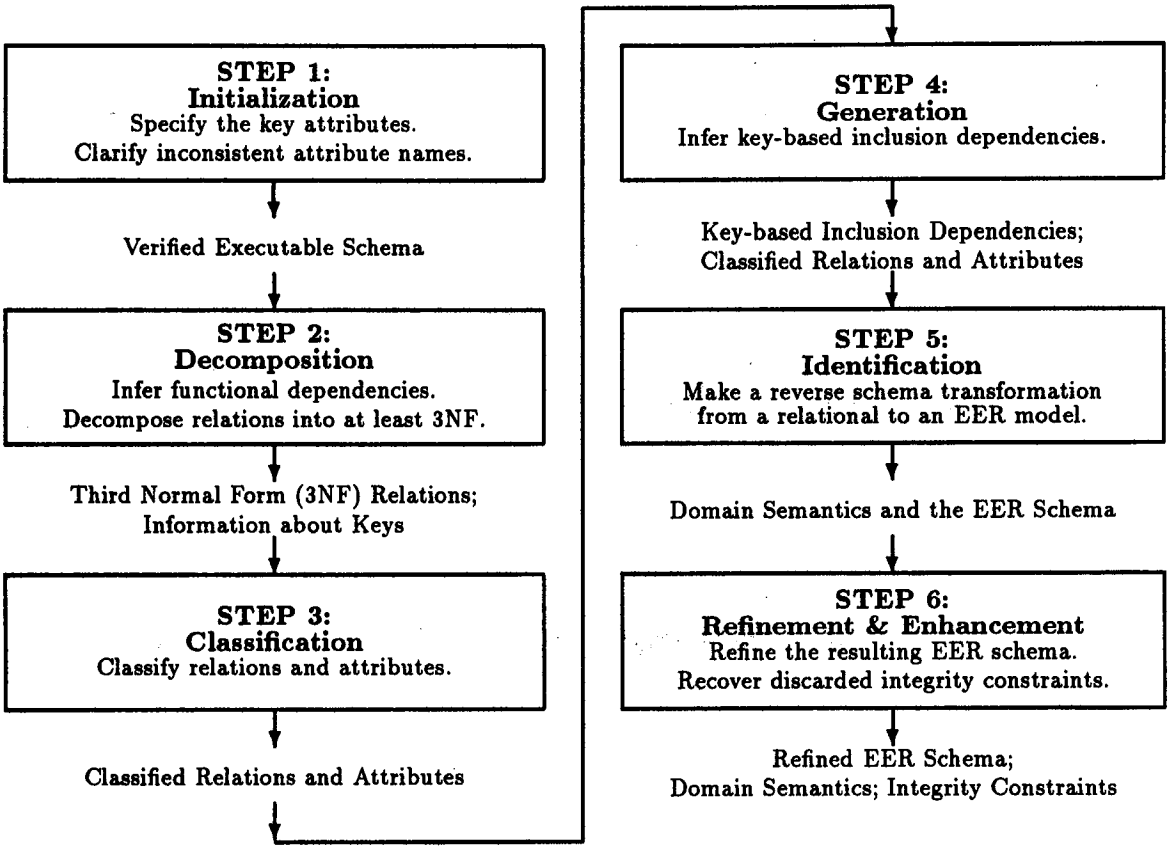

Figure 1: The Discovery Process

ID, #) to find possible key attributes for each relation. Information about possible key attributes is then provided to the user to aid in specifying the primary key of each relation.

Two types of naming problems are common: *homonyms* and *synonyms*. *Homonyms* occur when the same name is assigned to attributes with different properties; *synonyms* occur when different names are assigned to attributes with the same domain. The clarification rule analyzes attributes' properties for mismatches or similarities to help the user detect inconsistent situations. Mismatches occur when attributes with the same name have different data types and/or sizes; mismatches of two attributes indicate that they may be *homonyms*. Similarity arises when attributes with different names have the same data type and size, or even the same data instances. Two similar attributes may be *synonyms*. Since not all naming inconsistences can be detected automatically, the user must be informed of this phenomenon. After the similar and mismatched attributes are identified, the user is asked to confirm the results and perform any necessary renaming.

2.2 Decomposition of Relations

The discovery process decomposes non-third normal form (3NF) relations[1] of the input database into at least 3NF, which allows the identification step to deal with relations which primarily correspond to one entity type or one relationship type, rather than more than one entity type or a mixture of entity and relationship types.

[1]See Elmasri and Navathe [8] for the terminology of relational databases.

RELATION	ATTRIBUTES	TYPE
PERSON:	[SSN, name, address]	(STRONG)
CUSTOMER:	[SSN, custid, name, sex, credit]	(STRONG)
EMPLOYEE:	[SSN, salary, hired-date, sex, super-ssn]	(STRONG)
MANAGER:	[SSN, rank, promotion-date, deptno]	(STRONG)
DEPARTMENT:	[DEPTNO, dept-name, location, ordid]	(STRONG)
PRODUCT:	[PRODID, description]	(STRONG)
PRICE:	[PRODID, minprice, maxprice]	(STRONG)
MACHINERY:	[MACID, description, cost]	(STRONG)
PROJECT:	[PID, projname, budget, dept-name]	(STRONG)
DEPENDENT:	[NAME, SSN, sex, bdate, relationship]	(WEAK)
ORDER:	[PRODID, CUSTID, ordid, order-date, qty]	(REGULAR)
WORK-ON:	[SSN, PID, start-date]	(REGULAR)
SCHEDULE:	[MACID, ORDID, schedule-time]	(REGULAR)
EQUIPMENT:	[CARRIERID, PRODID, CUSTID, ship-date]	(SPECIFIC)
COMM-PROJ:	[PID, COMM-PID]	(SPECIFIC)

Figure 2: A Relational Data Schema

Primary keys and functional dependencies are needed to identify relations that violate the 3NF requirement. After non-3NF relations are identified, some standard algorithm (e.g., [3]) can then be used to decompose them into ones that satisfy the 3NF requirement. Previous research has studied how to infer functional dependencies by analyzing data instances, e.g., [1, 10, 11] using probabilistic methods, and at least one commercially available tool, DB Designer by Cadre Technologies Inc., uses these methods to infer functional dependencies. The data schema shown in Figure 2[2] is taken to be the executable schema resulting from these steps. This example is used throughout our paper.

2.3 Classification of Relations and Attributes

Each relation is classified based on the comparison of its key(s) with other relations' keys into one and only one of four possible categories: Strong Entity Relation, Weak Entity Relation, Regular Relationship Relation, and Specific Relationship Relation. The attributes of each relation are then classified depending upon their properties: 1) whether it is part of the primary key; 2) whether the non-primary-key attribute(s) are the key of another relation. Each attribute is classified into one and only one of five possible categories: Primary Key Attribute (PKA), Dangling Key Attribute (DKA), General Key Attribute (GKA), Foreign Key Attribute (FKA), and Non-Key Attribute (NKA). See Chiang et al. [7] for detailed discussion of the classification schemes of relations and attributes.

2.4 Generation of Inclusion Dependencies

An inclusion dependency is denoted as $A.X \ll B.Y$, where A and B are relations, X is an attribute or a set of attributes of A, and Y is an attribute or a set of attributes of B. X and Y must have the same number of attributes [4]. This inclusion dependency states that the set of values appearing in A.X must be a subset of the set of values appearing in B.Y.

Inclusion dependencies are generated by the following sequence. First, possible inclusion dependencies are formulated by the formulation rules. Second, invalid dependencies are detected and

[2]The names of relations and primary keys are shown in capital letters.

eliminated by the rejection rule. Finally, redundant dependencies are detected by the inference rule and then removed.

2.4.1 Formulation of Possible Inclusion Dependencies

In order to avoid formulating many inappropriate dependencies, the rules only propose inclusion dependencies between relations' key attributes (primary, candidate, and foreign). These rules are outlined below:

1. **IF:** two strong entity relations, A and B, have the same key, X,
   THEN: formulate A.X $\ll$ B.X and B.X $\ll$ A.X.
   JUSTIFICATION: Subtype/supertype relationships and vertically fragmented relations [8] result in such inclusion dependencies.

   Consider the strong entity relations PERSON and EMPLOYEE in Figure 2. The following inclusion dependencies are formulated:

   $$\begin{aligned} \texttt{PERSON.[SSN]} &\ll \texttt{EMPLOYEE.[SSN]} \\ \texttt{EMPLOYEE.[SSN]} &\ll \texttt{PERSON.[SSN]} \end{aligned}$$

2. **IF:** the key, X, of a relation (entity or relationship), A, appears as a foreign key, X, of another relation (entity or relationship), B,
   THEN: formulate B.X $\ll$ A.X.
   JUSTIFICATION: Cases where foreign keys represent binary relationships between entity types or between entity and relationship types result in such dependencies.

   Consider the relations MANAGER, DEPARTMENT, and ORDER. The following inclusion dependencies are formulated:

   $$\begin{aligned} \texttt{MANAGER.[deptno]} &\ll \texttt{DEPARTMENT.[DEPTNO]} \\ \texttt{DEPARTMENT.[ordid]} &\ll \texttt{ORDER.[ordid]} \end{aligned}$$

3. **IF:** the key, X, of an entity relation, A, appears as the primary key attribute(s) of a weak entity relation, W,
   THEN: formulate W.X $\ll$ A.X.
   JUSTIFICATION: The presence of weak entities' keys will result in this type of inclusion dependency. These dependencies are used to determine the owner entity types of weak entities.

4. **IF:** the key, X, of a relation (entity or relationship), A, appears as the primary key attribute(s) of a relationship relation (regular or specific), R,
   THEN: formulate R.X $\ll$ A.X.
   JUSTIFICATION: Relationships represented by relationship relations will result in such inclusion dependencies. These dependencies are used to determine participating entity types of relationship types identified by relationship relations.

2.4.2 Rejection of Invalid Inclusion Dependencies

Each proposed inclusion dependency is subject to further analysis to validate or reject it. Since erroneous data instances are to be expected, we use hypothesis testing to avoid rejecting dependencies that should hold but are violated in the database instance due to data errors. For each proposed inclusion dependency, A.X $\ll$ B.Y, two hypotheses are formulated:

H_0: A.X $\ll$ B.Y holds in reality.
H_1: A.X $\ll$ B.Y does not hold.

The parameters used are defined as follows:

r = true error rate in the data in question
a = number of data instances in A.X
b = number of data instances in B.Y
e = number of data instances, E, that appear in A.X but not in B.Y
$\hat{r} = \frac{e}{(a+b)}$

The reverse engineering process analyzes data instances of A.X and B.Y to obtain the values of a, b, and e. The null values in A.X and B.Y are not counted in a, b, and e.

Under the null hypothesis, $\hat{r}$ is a point estimator of the true but unknown error rate, r. (Some types of data entry errors will not be included in this, such as when data entered for A.X corresponds to an existing value of B.Y, but the wrong tuple is referenced.) A confidence interval of any desired confidence level can then be constructed around $\hat{r}$ using standard methods for estimating proportions (see [14], p. 196.) If the upper bound of this interval is larger than any error rate the user is willing to accept as plausible, then the null hypothesis is rejected; otherwise it cannot be rejected. (Clearly, if the user specifies $r = 0$, the presence of any erroneous data will cause the inclusion dependency to be rejected, so that error-free data is handled as a special case.) As is standard in hypothesis testing, a *type-one error* occurs when a valid inclusion dependency is rejected, while a *type-two error* occurs when an invalid inclusion dependency is not rejected. The output of the discovery process thus needs to be reviewed and verified by the user, and problems arising from such errors might be found then. Note that failure to reject an inclusion dependency, even when it is fully satisfied by the current database instance, does not guarantee that it holds in all possible database states.

:

2.4.3 Removal of Redundant Inclusion Dependencies

Redundant inclusion dependencies need to be detected and removed, because they can lead to redundant relationships, or identifying the wrong participating entity type for a relationship type, or both. This inference rule is used to detect redundant dependencies:

IF: A.X $\ll$ B.X and B.Y $\ll$ C.Y hold, and Y is a subset of X,
THEN: A.Y $\ll$ C.Y is redundant.
JUSTIFICATION: It is based on the projection and transitivity properties of inclusion dependencies.

2.5 Identification of Entity and Relationship Types

Identification rules encode knowledge necessary for performing reverse schema transformations from the relational data schema to the EER schema. Table 1 summarizes the reverse schema transformations based on classified relations and attributes. Identification rules are used to identify entity types first so relationship types among them can be identified thereafter. Several identification rules are discussed in detail below.

Table 1: Identification of EER structures by classified relations and attributes.

Relation	Attribute	Modelling Structures in the EER Model
STRONG		• **Identifies a strong entity type**, or • **Identifies a relationship type with its own key**
	PKA	Key for the strong entity type
	FKA	Identifies a binary relationship type
	NKA	Descriptive attribute for a relationship or an entity type
WEAK		**Identifies a weak entity type**
	PKA	Relates a weak entity type with its identifying owner(s)
	DKA	Key for the weak entity type
	FKA	Identifies a binary relationship type
	NKA	Descriptive attribute for a relationship or an entity type
REGULAR		**Identifies a relationship type**
	PKA	Determines participating entity types
	FKA	Identifies a binary relationship type
	NKA	Descriptive attribute for a relationship type
SPECIFIC		**Identifies a relationship type**
	PKA	Determines participating entity types
	GKA	Identifies new strong entity type(s)
	FKA	Identifies a binary relationship type
	NKA	Descriptive attribute for a relationship type

Strong Entities.

> **IF:** a relation, `A`, is classified as a strong entity relation,
> **THEN:** a) identify a strong entity type, **A**, and b) assign primary and candidate keys of `A` to **A**.
> **JUSTIFICATION:** According to Table 1, a strong entity relation can be converted into one of two EER structures. Since a relationship type with its own key can also be considered as an entity type, all strong entity relations are identified as strong entity types of the EER model.

For example, `CUSTOMER`, a strong entity relation, is converted into a strong entity type, **Customer**. The primary and candidate keys of `CUSTOMER` (i.e., `SSN` and `custid`), are assigned as the key attributes of **Customer**.

***Is-a* relationships.** *Is-a* relationships are identified using the following rule:

> **IF:**
> 1. two strong entity types, **A** and **B**, have the same key, X, and
> 2. `A.X` $\ll$ `B.X` holds, but `B.X` $\ll$ `A.X` does not,
> **THEN:** identify an *is-a* relationship between **A** and **B**.
> **JUSTIFICATION:** The first condition is the heuristic used to detect a subtype/supertype relationship between two strong entity types having the same key. The second condition employs an inclusion dependency to confirm the existence of an *is-a* relationship.

If two entity types, **A** and **B**, have the same key and the same set of data instances in their keys, then the user is asked to specify the proper type of *inclusion* relationship between **A** and **B**, such as **A** *is-a* **B**, **A** *is-a-kind-of* **B**, **A** *is-part-of* **B**, **A** *has* **B**, etc.

Relationships identified by relationship relations. The identification rule identifies a relationship type among the participating entity types for each relationship relation.

> **IF:** there is a relationship relation, R,
> **THEN:** identify a relationship type, **R**, with its participating entity types.
> **IF:**
> 1. the key, X, of an entity type, **A**, appears as a primary key attribute(s) of the relationship relation, R, and
> 2. R.X $\ll$ A.X holds,
> **THEN:** **A** is a participating entity type of **R**.
>
> > **IF:** more than one entity type, **A**, satisfies the above conditions,
> > **THEN:** the user must confirm the exact participating entity type.
>
> **JUSTIFICATION:** Conditions 1 and 2 above are necessary conditions for **A** to be a participating entity type of **R**. If **A** is not unique in satisfying these conditions, then further information which can only be supplied by the user is needed.

Consider WORK-ON in Figure 2. Suppose the following two inclusion dependencies hold:

> WORK-ON.[SSN] $\ll$ EMPLOYEE.[SSN]
> WORK-ON.[PID] $\ll$ PROJECT.[PID].

The reverse engineering process identifies a binary relationship type between **Employee** and **Project**, called WORK-ON.

Relationships identified by foreign keys. The identification rule identifies a binary relationship type for each foreign key.

> **IF:** a foreign key, X, in a relation (entity or relationship), B, appears as a key, X, of another relation (entity or relationship), A,
> **THEN:** identify a binary relationship type. One participating entity type is identified by the relation containing the foreign key (i.e., **B**). The other is identified by the relation whose key is the same as the foreign key (i.e., **A**).
> **JUSTIFICATION:** The foreign keys in the relational databases are used to represent one-to-many binary relationship types. Therefore, such an identification is the most appropriate transformation of a foreign key. The name of this relationship type must be provided by the user.

Consider the foreign key deptno in MANAGER and the inclusion dependency,
MANAGER.[deptno] $\ll$ DEPARTMENT.[DEPTNO]. The process identifies a binary relationship between the entity types **Manager** and **Department**, for which the user must provide a name, for example MANAGE.

2.6 Refinement and Enhancement.

The resulting EER schema is then refined and enhanced to make it semantically richer and more natural. First, the schema may contain EER structures that are identified by relational structures resulting from performance optimization (e.g., vertical fragmentation of relations). Second, design problems implicitly represented in the executable schema can be detected by the resulting EER schema. Finally, the resulting EER schema and the inclusion dependencies are used to declare integrity constraints for the existing database.

Key and Entity Integrity Constraint.

IF: X is a key (primary or candidate) of relation A,
THEN: X must contain unique values in any instance of A.
IF: X is also the primary key,
THEN: X must contain non-null values in any instance of A.

Domain and Referential Integrity Constraint.

IF: the key, X, of a relation, A, appears as either:
1. the foreign key, X, of another relation, B,
2. the key of a strong entity relation, B, or
3. the primary key attributes of a relationship relation, R,

THEN: (a) the foreign key X in B and/or the primary key attribute(s) X in R must have the same data type and size as X in A, and (b) each data instance of the foreign key X in B and/or the primary key attribute(s) X in B and R must refer to the data instance of X in an existing tuple of A.

3 The Knowledge Extraction System

The Knowledge Extraction System (KES), an interactive knowledge-based system, has been developed to perform the discovery process. KES illustrates how the integration of database and knowledge-based systems can support the work of knowledge discovery in relational databases. Currently, KES performs the last four steps of the discovery process.

KES is written in Arity Prolog and Microsoft C, integrated with the ORACLE RDBMS; it runs on an IBM PC. It has six basic components. Figure 3 shows the architecture of KES and the predominant information flows among its components. See Chiang [6] for detailed discussion of KES. Figure 4 shows the EER diagram representing the domain semantics discovered by KES from Figure 2.

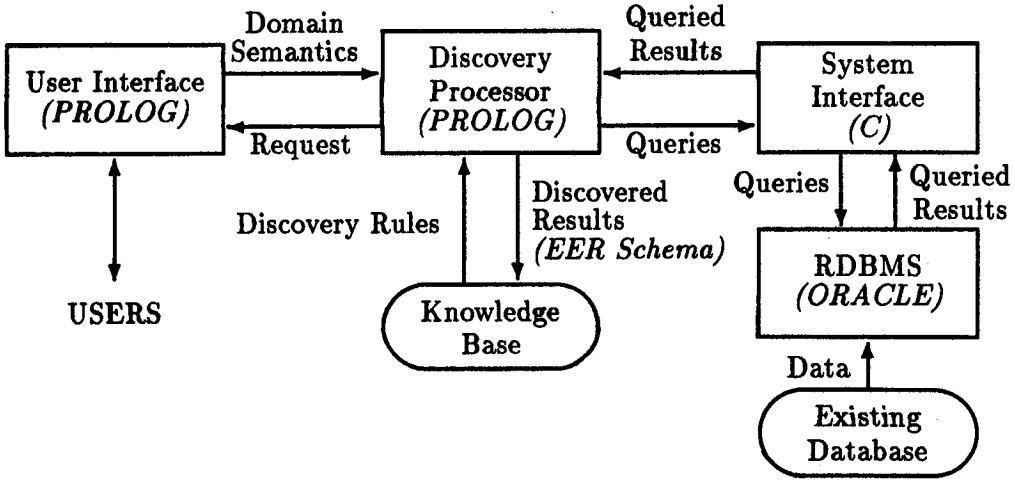

Figure 3: Architecture of the Knowledge Extraction System

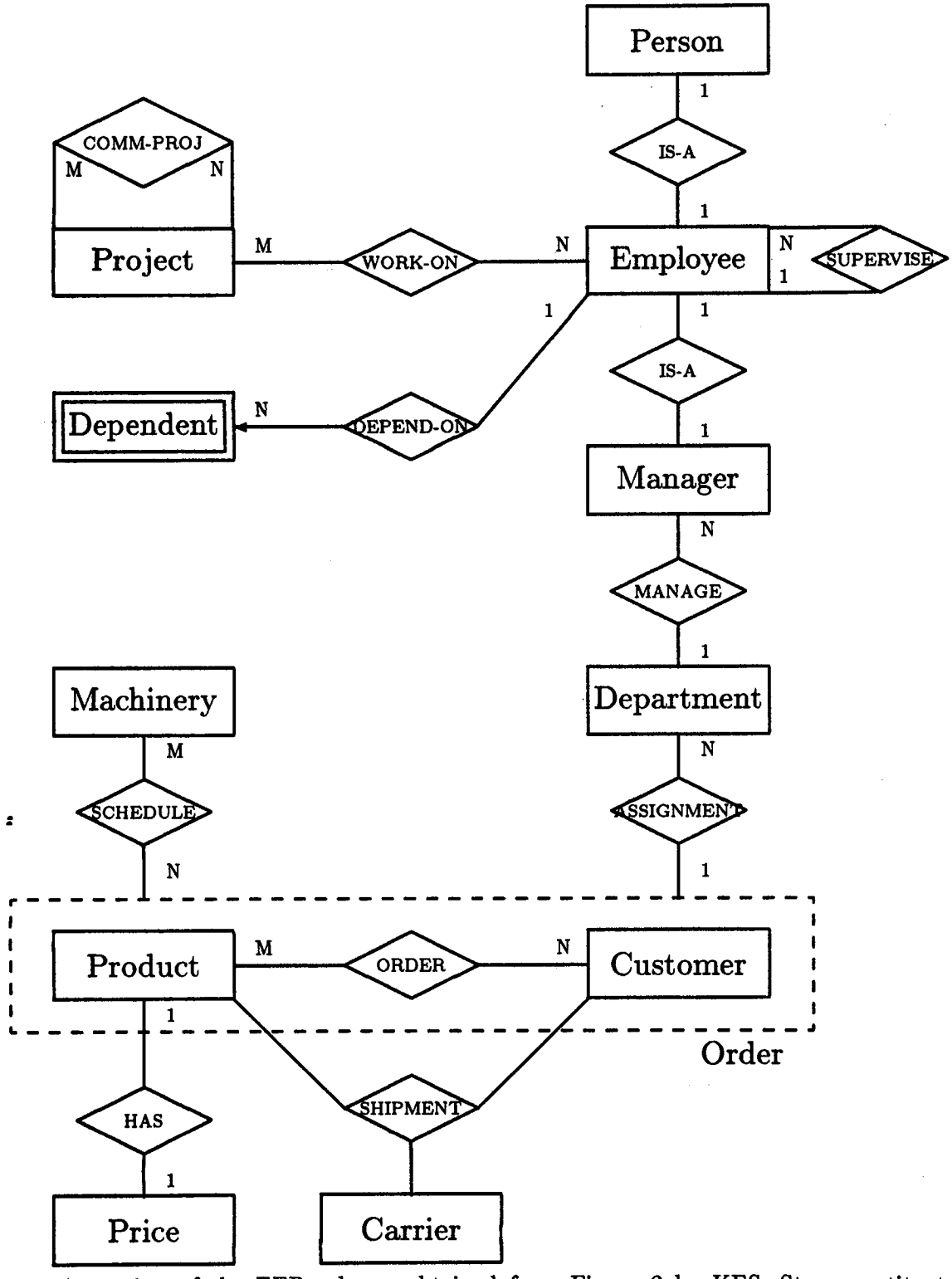

Figure 4: A portion of the EER schema obtained from Figure 2 by KES. Strong entity types are shown as rectangles, weak entity types are shown as double rectangles, relationship types are represented by diamonds, and composite entity types are represented by dashed boxes. The cardinality ratio of a binary relationship is shown by a pair of numbers along with the diamond.

4 Concluding Remarks

Our research investigates how original design specifications of an exiting relational database can be recovered by analysis of data instances and the executable schema. We present a process for discovering domain semantics for existing databases. These discoveries are represented in an EER schema that can provide a better interpretation of data with respect to the application domain in order to support subsequent knowledge discovery processes. Since the original design specifications are normally either unobtainable or uninteresting due to the various semantic degradations during the design and maintenance of a database, the EER diagram resulting from the discovery process should be considered as the best guess (most likely) design specifications as to what a rational designer would use the relational structures appearing in the database to represent. The discovery process uses discovery rules and the executable schema and data instances of the existing database, and it deals with the user if necessary. The Knowledge Extraction System, an interactive knowledge-based system, has been developed to demonstrate this discovery process can be implemented at a high level of automation with the application of knowledge-based systems technology. The results of our research should have immediate practical applications in research on knowledge discovery in databases.

References

[1] Akutsu, T. and Takasu, A., "Inferring Approximate Functional Dependencies from Example Data," in *Proc. of AAAI-93 Workshop on Knowledge Discovery in Databases*, The AAAI Press, July 1993, pp.138–150.

[2] Batini, C., Ceri, S. and Navathe, S.B., *Conceptual Database Design: An Entity-Relationship Approach*, The Benjamin/Cummings Publishing Company Inc., 1992.

[3] Bernstein, P.A., "Synthesizing Third Normal Form Relations from Functional Dependencies," *ACM Transactions on Database Systems* 1:4, December 1976, pp. 272–298.

[4] Casanova, M.A., Fagin, R. and Papadimitriou, C., "Inclusion Dependencies and Their Interaction with Functional Dependencies," *Journal of Computer and System Sciences* 28:1, February 1984, pp. 29–59.

[5] Chiang, R.H.L., "Reverse Engineering of Relational Databases: Extraction of Domain Semantics," unpublished Ph.D. thesis, University of Rochester, May 1993.

[6] Chiang, R.H.L., "A Knowledge-Based System for Performing Reverse Engineering of Relational Databases," *Decision Support Systems*, Forthcoming, March 1995.

[7] Chiang, R.H.L., Barron, T.M. and Storey, V.C., "Reverse Engineering of Relational Database of Relational Databases: Extraction of an EER Model From a Relational Database", *Data & Knowledge Engineering*, Vol 12, No, 2, March 1994, pp. 107-142.

[8] Elmasri, R. and Navathe, S.B., *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company Inc., Second Edition, 1994.

[9] Hainaut, J-L., "Database Reverse Engineering: Models, Techniques and Strategies," in *Proc. of the Tenth International Conference on Entity-Relationship Approach*, 1991, pp. 729–741.

[10] Kantola, M., Mannila, H., Raiha, K. and Siirtola, H., "Discovering functional and inclusion dependencies in relational databases," *International Journal of Intelligent Systems*, Vol. 7, 1992, pp. 591–607.

[11] Kivinen, J. and Mannila, H., "Approximate Dependency Inference from Relations," *Database Theory, ICDT92, Fourth International Conference*, 1992, pp. 86-98.

[12] Frawley, W.J., Piatetsky-Shapiro, G. and Matheus, C.J., "Knowledge Discovery in Databases: An Overview," *AI Magazine* 13:3, 1992, pp. 57–70.

[13] Premerlani, W.J. and Blaha, M.R., "An Approach for Reverse Engineering of Relational Databases," *Proc. of the Workshop on Reverse Engineering, 1993 International Conference on Software Engineering*, Baltimore, May 1993.

[14] Spiegel, M. R., *Probability and Statistics*, McGraw-Hill, 1975.