Tools and Transformations—Rigorous and Otherwise—for Practical Database Design

ARNON ROSENTHAL
The MITRE Corporation
and
DAVID REINER
Kendall Square Research

We describe the tools and theory of a comprehensive system for database design, and show how they work together to support multiple conceptual and logical design processes. The Database Design and Evaluation Workbench (DDEW) system uses a rigorous, information-content-preserving approach to schema transformation, but combines it with heuristics, guess work, and user interactions. The main contribution lies in illustrating how theory was adapted to a practical system, and how the consistency and power of a design system can be increased by use of theory.

First, we explain why a design system needs multiple data models, and how implementation over a unified underlying model reduces redundancy and inconsistency. Second, we present a core set of small but fundamental algorithms that rearrange a schema without changing its information content. From these reusable components, we easily built larger tools and transformations that were still formally justified. Third, we describe heuristic tools that attempt to improve a schema, often by adding missing information. In these tools, unreliable techniques such as normalization and relationship inference are bolstered by system-guided user interactions to remove errors. We present a rigorous criterion for identifying unnecessary relationships, and discuss an interactive view integrator. Last, we examine the relevance of database theory to building these practically motivated tools and contrast the paradigms of system builders with those of theoreticians.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*programmer workbench*; *software libraries*; H.2.1 [**Database Management**]: Logical Design—*data models*; *normal forms*; *schema and subschema*

General Terms: Design, Theory

Additional Key Words and Phrases: Applications of database theory, computer-aided software engineering, database design, database equivalence, data model translation, design heuristics, entity-relationship model, heuristics, normalization, view integration

Authors' addresses: A. Rosenthal, the MITRE Corporation, 202 Burlington Road, Bedford, MA 01730; email: arnie@mitre.org; D. Reiner, Kendall Square Research, 170 Tracer Lane, Waltham, MA 02154; email: reiner@ksr.com.

1. INTRODUCTION

The Database Design and Evaluation Workbench (DDEW) is a graphical database design system prototype built at Computer Corporation of America.[1] In this article, we describe how DDEW's facilities for conceptual and logical design weave together both formally justified and heuristic tools.

The DDEW tool suite supports multiple design methodologies, including synthetic and analytic techniques for design from scratch, reverse engineering of existing schemas, and pairwise integration of schemas. Conceptual design is done in the Entity-Relationship (ER) model, and the system handles multiple logical-level data models (relational, network, and hierarchical). This breadth of coverage brought with it two challenges: keeping behavior consistent and avoiding redundant development. Both representations and design operations exploit the strong similarities between the conceptual and logical models, and among the various logical models.

The user interface includes special features to help the design process. The designer sees a graphical view of the history and derivation of a given design in an on-screen *design tree*. Levels in the tree correspond to requirements and to conceptual, logical, and physical designs. Clicking on a node at a particular level opens the appropriate schema diagram. Additionally, several mechanisms are provided to help the designer to cope with complexity and with large designs. These include the use of color to identify missing or dubious information, highlighting of arbitrary *affinity groups* of related objects, and a miniature navigational aid for visualization of and movement within large designs. We also introduced a new graphic notation to show minimum and maximum participation in a relationship, without cluttering the diagram.

Early user reaction to the DDEW prototype was quite positive.[2] Its hierarchy of windows from the navigational aid and design tree on down helped the user to see the broad picture, to compare ER designs, and to deal with local details. The multistep methodology was seen as both correct and important, and the comprehensive and serious nature of the tool set was viewed as critical in a database design workbench. Users' favorite tools were the view integration tools and the ER schema synthesizer; these were viewed as providing extremely useful functionality. However, the transaction specification language was too low-level, additional report generation capabilities were requested, and there were some complaints about specific user interface features and performance on very large schemas (hundreds of entities).

For a broad overview of DDEW, including the user interface, graphical display of progress, design methodologies, and implementation considerations, see Reiner et al. [1986]. The present article concentrates on three of the unusual aspects of the system's data models and tools: (1) a unified underly-

[1]DDEW was supported by Rome Air Development Center, USAF, under contract F30602-83-C0073 to Computer Corporation of America. This article was written partly while A. Rosenthal was at ETH-Zurich.

[2]Initial users were from Argonne National Laboratories, Syscon, the Joint Analysis Directorate of the Pentagon, and the Carnegie-Mellon Software Engineering Institute.

ing data model, (2) content-preserving schema transformations, and (3) heuristic tools for initial design.

A major goal in writing this article was to illustrate the *interaction* of theory and practice. The data model, tools, and underlying transformations were considered successful because they simplified the design and implementation of a complex system which supports multiple data models and design methodologies. We aim at two audiences: builders of database design systems who might be tempted to utilize theory and theoreticians who want their work to be accessible to system-building teams.

1.1 A Unified Underlying Data Model

DDEW users see different data models, depending on the design stage and the target DBMS—an entity-relationship model for conceptual design and relational, logical, or hierarchical models at other stages. All of these are built over a single internal model, a simple ER extension (called *ER* + , created for this project), which provides the internal representation, transformations, and editing operations for designs. Thus, the same tool code runs under many different circumstances, maximizing consistency and minimizing the learning burden on users. Transformations are effectively shared among all of the models, making it possible to meet a very large number of system needs, while keeping model complexity and redundancy manageable.

1.2 Content-Preserving Schema Transformations

As a design progresses, details are accumulated, errors are fixed, and the schema gradually reorganized. Ideally, each change makes the schema a more accurate model of the external world. Once accuracy has been attained, however, conceptual and logical schema changes are generally still necessary for several reasons: *convenience* (so that schema objects will match organizational units or existing definitions), *implementability* (to use only the structures permitted in a target DBMS), and *performance* (since most DBMS's logical schemas partially determine physical implementation).

Schema transformations should not inadvertently introduce new errors into the modeling of the real world. Most database design systems and many published algorithms introduce unintended deviations, where designer-supplied constraints do not hold on the output of a transformation. This is a serious problem. If late transformations can introduce errors, correctness rests only on the designer's final check, instead of increasing throughout the design process. DDEW shows the feasibility, utility, and costs of a more rigorous approach: defining and exploiting transformations whose outputs are guaranteed to be content-equivalent to their inputs.

As formalized below, the *information content* of a schema is defined by the set of legal states of the database. Two schemas are called *content equivalent* if there is an invertible (total, onto, 1:1, and attribute preserving [Hull 1984]) mapping between their possible instantiations. A *rearrangement* is a transformation whose result is content equivalent to its input. The rearrangements in DDEW were of the following types: replicating the attributes of an

entity in related entities, and (inversely) eliminating such replication; converting a complicated relationship to an entity and two simpler relationships; inferring additional constraints, and (inversely) removing redundant constraints; creating keys; and replacing constructs not supported in a particular logical data model. The combination of rigorous specification and a unified data model enabled us to build many of our tools from a library of small rearrangement modules.

1.3 Heuristic Tools for Initial Design

Early in the design process, schemas are sketchy, omitting many relationships, attributes, and constraints. Content-preserving transformations that rearrange the early information will not yield an accurate "real-world" model, no matter how rigorously they are applied. Instead, *heuristic tools* are needed. These guess *possible* improvements based on built-in assumptions and defaults. The system then asks a designer to confirm proposed actions or provides visual cues about decisions that are considered doubtful.

Heuristic tools in DDEW support a number of major design processes: normalization during redesign of existing schemas, inference of relationships, detection of redundant relationships, and integration of conflicting user views. Errors from heuristic tools may be detected by human inspection, aided by tool display conventions that identify error-prone decisions. It is left to the designer to specify or select the corrections explicitly.

1.4 DDEW's Tool Suite

Figure 1 shows how DDEW's tools and transformations support design methodologies. Automated tools and their underlying transformations are listed, with references in square brackets to the sections describing the transformations. There were also a number of manual editing tools for lists and diagrams available throughout the design process.

DDEW supports several methodologies, including reengineering and integration of existing databases. For example, a designer working from scratch with functional dependencies might use the ER Schema Synthesizer to get the initial ER schema, the ER Schema Refiner to simplify it, the ER-to-Relational Translator, and various logical and physical design tools. A designer who started by entering an ER schema might run the ER Schema Refiner, the Normalizer, and various lower-level tools. A designer wishing to extend an existing relational or network schema might use one or both of the reverse-engineering tools to produce a conceptual schema, simplify it with the ER Schema Refiner, integrate it through the View Analyzer/Synthesizer tools with another ER schema derived from new requirements, and then continue with logical and physical design tools on the new schema.

The individual methodologies are straightforward and are not described in detail—the interest lies in how a relatively small set of modules was able to assist users in applying them. For detailed overviews of research-oriented and commercially oriented database design systems, see Ram [1992] and Reiner [1991]; also see Reiner [1992] for a summary of issues.

Step	Objects Accessed	Automated Tools Provided
Requirements Analysis	English description Functional dependencies (FDs) Transactions (against attributes)	**Cross-Reference Checker** (compares attributes in transactions with those in FDs)
Conceptual Design	FDs and transactions	**Entity-Relationship Schema Synthesizer [4.1]** (creates conceptual design from FDs: via normalization and entity creation [4.1], key creation [3.6], relationship synthesis [4.2] and constraint inference [3 4])
	ER schema diagram and transactions against it	**ER Schema Refiner** (attribute removal [3.3], transaction analysis [4 2 1 (f)]) **View Analyzer and View Synthesizer [4.3]** **Normalizer [4.1]** **Reverse Engineering tool** (logical --> conceptual translator· invokes relationship synthesis [4.2] and constraint inference [3 4])
Logical Design	ER schema diagram	**ER --> Relational Schema Translator [3.7.1]** **ER --> Network Schema Translator [3.7.2]** (attribute copying [3.2], key copying [3.2], relationship- to-entity conversion [3.5], relationship constraint inference [3.4], data model style transformations [3.7])
	Relational, network, or hierarchical schema diagram and transactions against it	**Logical Record Access tool** **Network --> Hierarchical Schema Translator [3.7.3]** **Schema Generators and Loaders** (Ingres, Troll/USE) **Reverse Engineering tool** (physical --> logical translator)
Physical Design	Relational, network, or hierarchical schema diagram augmented by physical specifications, transactions	**Physical Record Access tool** (index selection) **Schema Generator** (DMS/1100)

Fig. 1. DDEW's automated tool suite.

1.5 Organization of the Remaining Sections

Section 2 describes salient aspects of DDEW's unified underlying data model and explains how it was shaped by the need to support multiple target models and rigorous rearrangements. With this basis, Section 3 describes content equivalence, rearrangements for conceptual design, and mapping to a logical data model. Section 4 discusses how heuristics for normalization, relationship refinement, and view integration are used to obtain good starting schemas for further design. Section 5 comments on the contributions and limitations of the theoretical literature as it related to building DDEW, focusing on questions of comprehensibility, robustness, and graceful degradation. Section 6 gives conclusions. Preliminary versions of this work appeared in Reiner et al. [1986] and Rosenthal and Reiner [1987; 1989].

2. THE ER+ DATA MODEL

A *data model* is a set of constructs for expressing how data is structured, constrained, and manipulated. Different data models are required for different types of designs: conceptual schemas, logical schemas (suited to a particu-

lar data model or DBMS interface), and physical schemas that capture implementation detail. A database design system needs to represent and manipulate all of these, either as separate models or as special cases of a more general model.

We took the latter approach and found substantial benefits. One model, called ER+, provided the *internal representation and semantics* for all conceptual and logical design activities. The constructs and operations in ER+ were obtained by taking the union of all features in the four visible models (ER, relational, network, and hierarchical) and removing duplication. As a result, the same tool code could be run under many different circumstances.

Below, Section 2.1 presents the model, and Section 2.2 describes the use of model subsets for schema conversion. Section 2.3 explores the benefits and costs of building over a unified underlying data model.

2.1 ER+ Description

ER+ is in the Entity-Relationship family of data models. Its constraint constructs are numerous and rather general; they constitute a minimal collection that encompasses the constraints of the classical data models and allows our schema transformations to preserve information content. In models that do not express constraints, content-preserving transformations can do no more than (trivially) reorder attributes [Hull 1984].

ER+ begins with conventional entities, attributes, and relationships (binary, without attributes). If time had been available, we would have included generalization hierarchies, repeating groups, and possibly k-ary relationships and attributes on relationships. ER+ operations [Reiner and Gonzales 1985, Sections 6.2 and 7.2] provided a general but primitive means for specifying transactions. Of the constructs below, only value-determined relationships and the treatment of null values are unusual.

Let A and B denote lists of attributes, for entity types E1 and E2. The corresponding projections are denoted E1[A] and E2[B], respectively. e1[A] denotes the tuple of attribute values in A for an instance e1 of E1; e2[B] is defined analogously; pairs of attribute lists correspond by position rather than by name. R denotes a relationship between entities E1 and E2. If R includes the pair of entity instances (e1, e2), we say e1 and e2 are *R-related.* We present the most interesting constraints first.

Value-Determined Relationship. To bridge the gap between ER and relational models, we allow a relationship to be determined by matches of attribute values. A value-determined relationship represents a join path that is either semantically interesting or else is useful as a carrier of constraints. Formally, R is *value determined* by matching attributes A and B if R consists exactly of the set of entity pairs (e1, e2) such that e1[A] = e2[B].

Value-determined relationships subsume foreign-key references and provide several additional conveniences. First, they allow a join path to be associated unambiguously with a relationship name from the conceptual level and to have attached constraints (e.g., inclusions, participations, and even

English text). Also, the connection to particular attributes partially captures the relationship's semantics. This understanding can help reattach a relationship when normalization splits an incident entity (the relationship follows its determining attributes) and can help test whether one relationship is the composition of two others (see Sections 4.1 and 4.2.2).

Example. The value-determined relationship WORKS_IN between EMP and DEPT consists of {(e, d) | e[Dept#] = d[Dept#]}. Speaking loosely, we say that the "same" attribute Dept# appears in both entities; actually, it is values of the attribute that are replicated.

Example. Relationship MANAGES is determined by matching EMP[E#] = EMP[Manager#].

Null-Not-Allowed. Tuple t has a null value for attribute set A if every attribute in A is null, i.e., t[A] is nonnull if *at least one* attribute is nonnull. The null-not-allowed constraint, applied to sets of attributes A, requires that t[A] be nonnull.

Key. A is a key of E1 if and only if the values of attributes in A, if nonnull, uniquely identify the instance of E1. One key of E1 whose attributes are null-not-allowed may be declared *primary*. We allow primary keys to include null-allowed attributes, as long as the entire key still provides unique identification, and no subset does so.

For example, consider a PART relation where ordinarily Part# (which is null-not-allowed) suffices to identify a part, but some parts have multiple variants. The primary key might then be [Part#, Variant#]. Formally this problem can be resolved by splitting into separate entity types, but such a split adds complexity to the schema.

Inclusion. E1[A] *includes* E2[B] (denoted E1[A] $\supset$ E2[B]) if every nonnull value e2[B] appears as the value of some e1[A].

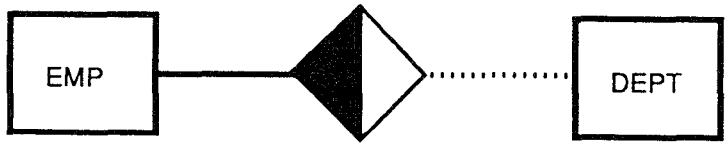
Example. DEPT[Dept#] $\supset$ EMP[Dept#] implies that the Dept# value associated with an employee must also be associated with a department.

Minimum and Maximum Participation. E1 has minimum (maximum) participation of k in R if each e1 is related to at least (respectively, at most) k instances of e2. E1 has *mandatory* participation in R if its minimum participation is greater than 0.

The notation "E1—(m1, M1)—⟨R⟩—(m2, M2)—E2" is used in this article to show (min, max) participation by the entities on each side of relationship R. Ignoring minimums, we loosely say that R is M2:M1. "Unknown" is a legal value for min and max participation values.

In DDEW's screen displays, we introduced two new easy-to-visualize participation notations. Maximum participations are displayed by splitting the diamond in half and coloring the halves (see example below). Each half represents the maximum participation of the connection with the closer object and can be colored white (1), green (many), or red (unknown). Since

minimum participations (i.e., minimum number of relationship instances in which an entity participates) are seldom different from 0 or 1, we represent them by a dotted green line (0), a solid green line (1), or a solid red line (unknown). DDEW highlights places were more information is needed, yet allows design to proceed based on partial information.



Our display notations have several useful properties and are particularly suited to large diagrams. They have little visual clutter. All participation information can be seen in one place (near the diamond); additionally, minimum participation is visible all along the lines.[3] "Unknown" is conveniently represented. Finally, we conserve space near the entity nodes (unlike multiple-arrow or crows-foot notations) and thus can display a large number of incident relationships. Our display notations are used (with adaptations) in Teorey [1990].

Set-Key. This construct models uniqueness constraints within a Codasyl set. Suppose each e1 of E1 is R-related to exactly one instance e2 of E2, and that e1[A] uniquely identifies e1 among the set of E1 instances related to e2 (but uniqueness is not guaranteed among *all* E1 instances). Then E1[A] is called a set-key of E1 with respect to R.

Relationship Parent. When creating a hierarchical schema from a network schema, the designer may declare which entity incident to a 1:1 relationship should be treated as the *parent*. For a 1:n relationship, the *parent* is the entity on the 1 side.

Constraints as Logic Formulas. It is natural to ask what role a general language like logic should play in expressing constraints. A database design system ought to be able to capture constraints beyond the small list above, in both English (for documentation) and logic (for evaluation, for determining which objects are constrained, and for manipulation by a theorem prover). However, the specific constructs above are still needed, to supply natural higher-level concepts. Both users and transformation implementors find a declaration "Key(E, K)" easier to handle than the equivalent logic assertion: "Instance(e, E) and Instance (e′, E) and equal(e[K], e′[K]) ⇒ equal(e,e′)." Even if transformations are performed in logic, a theorem prover must rewrite the output using constructs that users understand.

2.2 Using Style Subsets of ER+ for Schema Conversion

Ideally, a database design system should capture, display, and transform schemas in multiple models. Multimodel support widens a product's market,

[3]Participation values other than 0, 1, and, n can be viewed by zooming in on the relationship icon.

helps users to combine information from multiple sources, and—if done in an open fashion—promotes further extension and customization. In our case, the client specified that we support relational, network, and hierarchical models for the logical DBMS interface, and an extended ER model for conceptual design. The industry has changed since DDEW was completed, but support for network and hierarchical models continues to be important for reengineering.

In practice, there is often a need for multiple models at the conceptual as well as the logical level. A large organization might want support for several ER variants (e.g., from different CASE systems). In the future, object-oriented models with support for inheritance and data abstraction will be increasingly important. In Section 2.3.4 we examine how DDEW's approach (unified model and reusable transformations) might fare if the system were extended to better support such models.

A *style subset* of ER+ is a set of ER+ constructs that corresponds to constructs appropriate for a target logical model. In the *network* style, relationships are 1:1 or 1:n, and they must connect distinct entity types. (Repeating groups were outside our scope.) The *hierarchical* style is considered a subset of the network style, with the additional conditions that all relationships have an identified parent entity (with participation mandatory for the child entity), that no entity can be related to more than one parent, and that there can be no relationship cycles of any length. In the *relational* style, all relationships must be value determined; every entity must have a primary key; and set-key declarations are prohibited. Note that relationships in the relational style correspond to matching sets of attributes (i.e., join paths) in the relational schema. Retaining them shows important connections and participation constraints that cannot be seen in a simple list of relations.

Style subsets of ER+ for relational and network models allow a two-step, conceptual-to-logical model conversion rather than the more customary direct, single-step translation. As illustrated in Figure 2, the first step in converting a schema to a target DDL is to rearrange the schema to the chosen style subset. This rearrangement may change the schema significantly (e.g., migrating attributes and converting relationships to entities); theory and code developed for other ER+ purposes do most of the transforming.

The second step is to perform a *purely syntactic* translation to the target DDL. We designed the style subsets to minimize the distance covered by this mapping. Constructs that cannot be translated are converted to *comments* on the resulting schema and need to be enforced by application programmers. In translating from relational-style ER+ to relational DDL, constructs converted to comments include relationships (now carried by attribute pairs), relationship names, and minimum and maximum participation (which are partly enforced by key and inclusion constraints). We do not address the myriad approaches to enforcing referential integrity [Markowitz 1990].

Our network and relational style subsets are *fully expressive*, in the sense that for *any* ER+ schema there exist equivalent network-style and relational-style schemas (see Section 3.7). The hierarchical-style subset *is not*

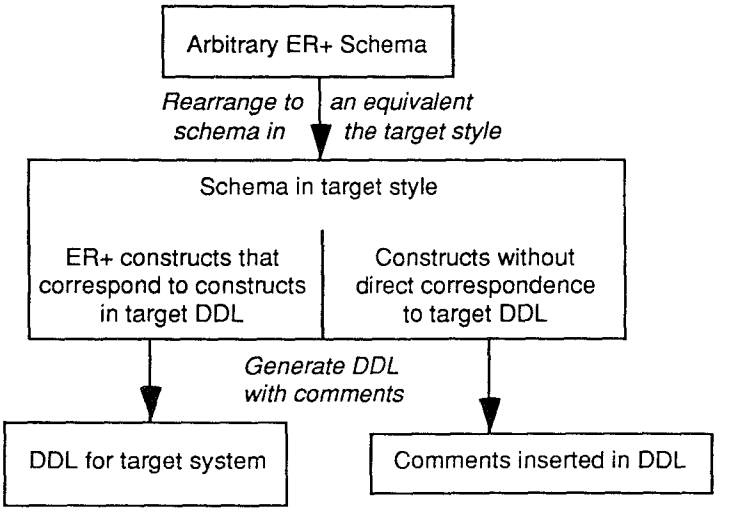


Fig. 2. Two-step data model conversion.

fully expressive, since equivalence can be lost when multiparent entities are reduced to single-parent ones or when cycles are arbitrarily broken (as was done in DDEW).

2.3 Benefits of a Unified Underlying Model

All significant schema modifications in DDEW are expressed within the ER+ formal system, which can be regarded as providing a collection of abstract data types, to be used in implementing the various user-visible models. Transformations rearrange information in an ER+ schema; import/export utilities and several other tools perform syntactic translations and little else. Building in this way over ER+ brought considerable benefits, as outlined below.

2.3.1 *Reusable Constructs and Operations.* Major rearrangements, both within and among style subsets, were written as compositions of small ER+ rearrangement modules. Nearly every ER+ feature was reused in several contexts, and such reuse substantially reduced development effort. We reused data model *constructs* (data structures and associated editing and display code) and schema *transformations* (code and detailed theory).

Reuse of Constructs. *Entities* in ER+ are used to implement conceptual entities, logical relations, logical entities (for an ER-based DBMS interface, as in the IRDS standard), and Codasyl record types, all of which involve an aggregate of attributes. *Relationships* in ER+ provide an underlying abstraction for semantic connections and for attaching constraints; IS-A connections could be implemented over relationships (see Section 3.8). *Attributes* are defined in all models. *Constraints* and *datatype declarations* appear at multiple levels.

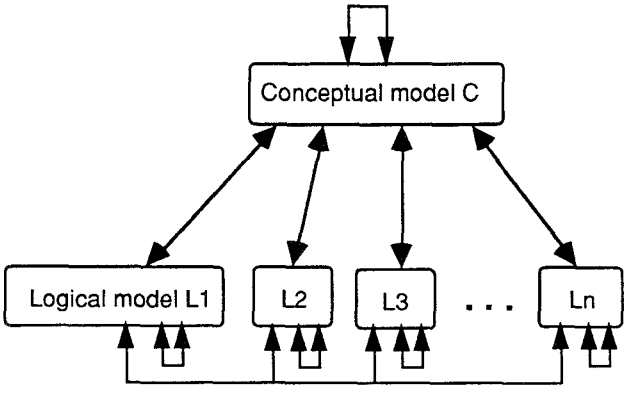


Fig. 3. Possible transformations without a unified underlying model.

Reuse of Transformations. Copying attributes from one entity into a related entity is important for generating keys in intermodel translation and for producing useful redundancy in a physical or logical schema. It would also be useful for forming an entity (or logical relation) that includes both local and inherited attributes of a subclass. The reverse operation (deleting a redundant attribute) is relevant when deriving a conceptual schema or when reversing a decision to store an attribute redundantly. Another repeatedly used ER+ operation replaces a relationship by an entity and two new relationships, adjusting constraints to preserve information content.

In a system with a single underlying model, one implements each transformation once. In contrast, Figure 3 shows the spaghetti-like potential for redundancy and inconsistency with independent models.

An alternative would be to translate all schemas into the relational model, transform then according to existing theory, and then translate them back. Translation into the relational model would lose ancillary information about a design (e.g., relationship names, diagrammatic layout, and textual annotations). It is quite difficult to recover this information for the transformed schema or to specify how it should be carried through relational transformations (as discussed briefly in Section 4.1.2).

Opportunities for reuse are limited—but still real—even in a more narrowly targeted workbench. Markowitz and Shoshani [1989] describes a workbench with a single extended ER conceptual model, no conceptual-to-conceptual transformations, and a robust flavor of relational as the only logical model.[4] This was accompanied by an extensive theory about correspondences between the conceptual and logical levels, for both constructs and operations. For logical-level equivalence and normalization the existing relational theory is used. Even in this situation, however, reformulating intermodel correspondences over a single formal system would factor out

[4]The "stricter" relational model style of Markowitz and Shoshani [1989] can be captured by imposing further constraints that all attribute values are nonnull and every value-determined relationship includes a key of at least one entity.

operations (such as attribute copying and deletion) that appear in both intermodel and intramodel transformations.

2.3.2 *Simplicity of Export/Import Utilities and Other Syntactic Transformation Tools.* A database design system requires numerous schema export and import utilities for its multiple target DBMSs. To generate a SQL variant (such as Ingres DDL) directly from an ER schema, one needs code to migrate key attributes and create relations for m:n relationships and appropriate constraints. However, the export utilities themselves should be straightforward translators with no responsibility for improving design decisions in their inputs. Hence, we first produce an ER diagram in the target style subset (see Figure 2) and then restrict the export utility to a purely syntactic translation.

Conversely, for tools that import foreign DDL we try to map each foreign construct directly to an ER+ construct (and ER+ has received extensions such as set-keys to permit this). If a source schema has redundancies, is missing constraints, or is otherwise "bad," it is still brought directly into ER+ by the import utility. A single ER+ tool ("Schema Refiner") then detects problems and improves the schema.

Similarly, the View Synthesizer tool is responsible for obtaining a semantically correct schema, but not for cleaning it up. The Normalizer tool just creates a relational schema, represented as a set of entities with no relationships.

2.3.3 *Flexibility in Information Capture.* Our modeling approach aims to provide flexibility, since we cannot anticipate future design methodologies. In some modeling approaches, information assigned to one phase of design (e.g., detailed constraints on data values, as part of logical design) cannot be captured at other phases. In contrast, common ER+ underpinnings allow the same schema to contain information typically associated with different design phases.[5]

Consequently, methodologies supported by DDEW may differ in the information assigned to each phase. Also, users who discover some important information "prematurely" (e.g., from reading existing documentation) may be allowed to enter that information. This is preferable to making multiple passes with the information source or to capturing information outside of the design system.

As mentioned in the previous section, ER+ has the flexibility to express "bad" schemas. Schemas directly imported from existing relational databases or flat files may violate recommended design practices, but may still be the best starting point for redesign.

Another benefit of the common underpinnings is that one can incrementally translate a schema between levels. This avoids a sudden jump into a logical design which unintentionally embodies physical decisions (since most DBMSs directly implement tuples as physical records). As another example,

[5]We did not encourage DDEW's end users to exploit this flexibility—each phase's user interface hid inappropriate information or placed it in a more detailed pop-up window.

for a new style subset corresponding to object-oriented DBMS, a translator that for some reason could not handle a construct newly added to ER+ may pass the offending information unchanged into the ER+ result.

2.3.4 *Complexity of the Unified Model.* The unified data model had a price —*complexity*. And future extensions to the data model could exacerbate the problem. Such extensions might include support for generalization hierarchies, constructs for non-first-normal-form databases, k-ary relationships, and physical storage and indexing constraints. We used several tactics to deal with this complexity, involving both the user interface and the design tools of DDEW.

(1) DDEW customizes its logical design displays to avoid redundant information. For example, the network model builds in the assumption that a relationship's child instance participates with at most one parent, so graphical displays need not show maximum participation. Schema details are generally viewed graphically within pop-up windows that limit and focus the information seen by the designer. Mantha [1987] carries these ideas further, allowing the user to dynamically adjust the information displayed.
(2) Some transformations apply only to style subsets, so before using the transform a user is constrained to first produce a schema in that subset. For example, generators of INGRES DDL will translate only constructs that belong to relational-style schemas.
(3) We made it possible for DDEW implementors to add constraint types to the model without invalidating existing tools. Tools that detect an unfamiliar constraint (either a new construct or a logic formula) are prevented from changing the object (entity, relationship, or attribute) that the constraint references. For example, attribute removal (see Section 3.3) will remove apparently redundant attributes only if the attributes are not mentioned in other constraints. Thus, tools degrade gracefully on schemas that contain a few unfamiliar constructs.

3. REARRANGEMENT TRANSFORMATIONS IN DDEW

A *rearrangement* is a transformation that preserves the information content (defined below) of a schema. DDEW demonstrates that a database design system can be built largely from modular rearrangements and that these modules may be reused in different contexts and for several model styles. Since real schemas often violate simplifying assumptions in published algorithms, our transformations *do* handle null-allowed attributes, multiple keys in an entity, and allow a pair of (not necessarily distinct) entities to be connected by several relationships. Our large tools were built as sequences of these independently verified rearrangements, each of which was a content-preserving adaptation of a familiar transformation.

Each of the transformations is described below for a single relationship and its incident entities. In general, a DDEW user command applies a

transformation to all relationships in the design and performs several different transformations in sequence. A transformation may include preconditions; no change occurs if these preconditions fail.

Section 3.1 formalizes the notion of content equivalence. Sections 3.2–3.7 show rearrangements that were useful in DDEW. These include attribute copying, attribute removal, inferring constraints for value-determined relationships, creating primary keys, transforming a relationship into an entity, and data model style translations. All the transformations described have been implemented; however, the descriptions incorporate improvements added afterward. Section 3.8 discusses extending the ER+ infrastructure to support inheritance.

3.1 Formalization of Content Equivalence

This section first presents an informal example of a transformation in which constraints must be introduced in order to make the output equivalent to the input. We then formalize the definition of content equivalence.

Example. Consider the familiar transformation that transforms an information-bearing relationship (in an ER schema) to a value-determined relationship (suitable for a relational system).

Before:	
Entity EMP(Emp#,Name,Address);	Emp# is primary key of EMP
Entity DEPT(Dept#,Dept_Name,Manager,Budget);	Dept# is primary key of DEPT
Relationship EMP_DEPT, with participations:	EMP — (1,1) — ⟨EMP_ DEPT⟩ — (1,N) — DEPT
After:	
Entity EMP′(Emp#,Name,Address,Dept#);	Emp# is primary key of EMP′
Entity DEPT(Dept#,Dept_Name,Manager,Budget);	Dept# is primary key of DEPT
Relationship EMP′_DEPT, with participations:	EMP′ — (1,1) — ⟨EMP′_ DEPT⟩ — (1,N) — DEPT

Additional Constraints

(1) EMP′_DEPT is value determined by matching Dept#, that is, EMP′_DEPT = {(e,d) | e ∈ EMP′, d ∈ DEPT, e[Dept#] = d[Dept#]}

(2) EMP′[Dept#] is nonnull

(3) DEPT[Dept#] ⊃ EMP′[Dept#]

In some papers and systems, the three additional constraints are not generated; yet without them it is possible to describe situations that do not hold in the original schema.

Two schemas are content equivalent if there is a natural 1-1 correspondence between the database populations that instantiate them. We now formalize this notion. An *entity scheme* is a set of attributes; a *relationship scheme* is a pair of entity names; a *schema* is a collection of entity and relationship schemes and a set of constraints. An *instantiation* of an entity scheme is a set of tuples of attributes; an *instantiation* of a relationship scheme is a set of pairs of entity instances. A *schema instantiation* is an

instantiation of all the entity and relationship schemes, such that the constraints are satisfied.

Let S1 and S2 denote schemas. A function from {instantiations of S1} to {instantiations of S2} is called an *instance mapping*, denoted I( ). I( ) is an *equivalence* if it is invertible and "natural." That is, for instantiations s1 and s2 of S1 and S2 respectively, I is:

(1) *total* (for each schema instantiation s1 of S1, I(s1) is an instantiation of S2)
(2) *surjective* (for every s2, there exists *at least one* instantiation s1 of S1 such that s2 = I(s1))
(3) *injective* (for each s2, there is *at most one* s1 such that s2 = I(s1))
(4) *natural*—attribute values are preserved by the mapping. The technical definition (called "generic" in Hull [1984]) is that the transformation is invariant under permutation of nonnull attribute values. Without this condition, every schema S1 would be equivalent to a degenerate schema consisting of a single integer attribute taking values between 1 and the number of instantiations of S1.

A schema transformation T is *content preserving* (i.e., a *rearrangement*) if for every schema S, there is an equivalence mapping to T(S). That is, the set of instantiations of S and T(S) are "equivalent." Because the composition of rearrangements is a rearrangement, rearrangements make good modules. (See Appendix for all proofs.)

The theory of rearrangements is very helpful, but its limitations must be understood. Content equivalence does not imply that design semantics (i.e., natural mappings to the real world) are preserved. For example, a rearrangement would be a disaster if it replaced every entity name by a unique integer or if it deleted unconstrained relationships whose population is value-determined by matching attributes.

3.2 Attribute Copying Transformations

A set of attributes can be copied from one entity E1 into an R-related entity E2, with constraints adjusted. This transformation has several uses in inter-model translation and physical design. To express an information-bearing relationship R in a relational-style ER+ schema, one typically imports (i.e., copies) a foreign key into E2 (see Section 3.6). To speed queries that need the attributes of E2 plus a few attributes of E1, one may denormalize—copying the desired attributes into E2.

The added constraints below assure content equivalence and prevent anomalies when updating a denormalized schema. The designer can then allow performance and interface convenience to govern which schema ought to be used. We restrict our attention to cases where the constraints on the resulting schema are expressible with ER+ constructs. The preconditions exclude situations where ER+ constraints (excluding logic) are not sufficient to obtain equivalence, such as copying *part* of a key, or copying across m:n

relationships. For simplicity, we assume that attribute names are unique; the actual DDEW implementation detects conflicts and generates new names.

3.2.1 *Key Copying.* An entire key may be copied to make a relationship value determined.

—*Copying a key* A1 *of* E1 *across a non-value-determined relationship* R *from* E1 *to* E2:

Preconditions for Applicability. R is not value-determined and max_participation(E2,R) = 1 (i.e., each e2 is R-related to at most one instance of E1). A1 contains a key of E1 (i.e., is a key or a superset of a key), and E1[A1] is null-not-allowed.

Result. Attributes A1 are added to E2. R has the constraint that it is value determined by A1. E1[A1] ⊃ E2[A1]. If R was mandatory from E2, then E2[A1] is null-not-allowed.

3.2.2 *Additional Attribute Copying*

—*Copying attributes A3 across a value-determined relationship R from E1 to E2:*

Preconditions for Applicability. R has at most one E1 for each E2 and is value-determined with E1[A1] matching E2[A2]. A1 contains a key of E1.

Result. E2 has the additional attributes A3; E1[A1 ∪ A3] ⊃ E2[A2 ∪ A3].

Example. Suppose we wish to copy the additional attribute Dept_Name from DEPT into EMP1 entities. The result is an entity type EMP2 that includes the Dept_Name in addition to other attributes of EMP1. The inclusion constraint is modified to be DEPT[Dept#,Dept_Name] ⊃ EMP2 [Dept#,Dept_Name].

3.3 Attribute Removal

Attributes that are determined by values in a related entity are deleted. The *Key removal* and *Additional attribute removal* transformations are the inverses of *Key copying* and *Additional attribute copying*, respectively, and will not be stated formally.

Applicability conditions are stringent—attribute removal applies only to schemas that could have been produced by attribute copying. This means that (1) there can be no constraints on the attributes to be removed, except for those imposed by copying and (2) the entity from which attributes are removed must be related to exactly one instance of the other entity. Postconditions of each form of attribute copying become preconditions of the corresponding attribute removal.

Redundant nonkey attributes arise when one reverse-engineers nonnormalized schemas and may be created also as a byproduct of integrating separate user views. Foreign-key references will be ubiquitous when one reverse-engineers a relational schema by heuristically inferring relationships (Section 4.2). Attribute removal is typically invoked after inclusion- and functional-dependencies have been inferred heuristically or manually added by the user.

3.4 Inferring Constraints for Value-Determined Relationships

The rearrangements in this section, invoked from several tools, straightforwardly infer constraints that are implied by other constraints. The inverse rearrangements remove constraints to produce a minimal set. Minimal sets impose less run-time overhead and are sometimes easier for other transformations to handle.

In the rules below, R12 denotes a *value-determined* relationship between E1 and E2, determined on matching E1[A1] and E2[A2].

—*Inferring maximum participation*:

If A1 contains a key of E1, impose the (maximum participation) constraint that at most one E1 instance participates in R12 for each E2. For example, suppose ⟨WORKS_IN⟩ relates EMP and DEPT (based on matching Dept#), and Dept# is a key of DEPT. Then there can be at most one ⟨WORKS_IN⟩-related DEPT instance for each EMP instance.

—*Inferring inclusions and null-not-allowed*:

If membership in R12 is *mandatory* for E2 instances, then E1[A1] ⊃ E2[A2], and E2[A2] is null-not-allowed. For example, if each Employee *must* be related (via WORKS_IN) to a department, then every Dept# value appearing in an EMP instance must be the number of a DEPT within this database.

—*Inferring minimum participations*:

Suppose that instead of specifying nonzero minimum participation, the user had specified null-not-allowed and inclusion constraints, i.e., that EMP[Dept#] is nonnull and DEPT[Dept#] includes it. Then the system could infer that each EMP must be ⟨WORKS_IN⟩-related to a DEPT. Formally: If E1[A1] ⊃ E2[A2] and E2[A2] is null-not-allowed, then minimum participation of E2 in R12 is at least 1. If the user *explicitly* allowed nulls for E2[A2], then minimum participation of E2 in R12 must be 0.

Note that although the "minimum participation = 0" declaration imposes no constraint, DDEW prefers explicit denials to omission—the denial is evidence that the question has been examined. In fact, when participation information has not been supplied, the corresponding half of the relationship diamond is displayed in an alarming red. As a heuristic, our transformations propagate "unspecified" values into their result.

3.5 Rearrangements Involving Key Creation

When files or Codasyl databases are being reverse-engineered, DDEW's direct import may create entities without keys. The following transformations supply keys, thereby enabling the entities to be treated within the relational model.

—*Creating a primary key for a keyless entity **E2**:*

Suppose that E2 has minimum_participation = maximum_participation = 1 in R12. And suppose E1 has a primary key K1 that can be copied

across relationship R12 to E2 (using the *key-copying* rearrangement of Section 3.2).

{If E1 has a maximum_participation = 1 in R12, then
(1) Copy K1 into E2 and declare copied attributes K2 a key of E2;
else If R12 has a null-not-allowed set-key SK, then
(2) Copy key K1 of E1 into E2 (as K2) and declare (SK ∪ K2) a key of E2,
else
(3) create a surrogate key for E2 as described below}

To illustrate case (1), suppose E1 is EMP and E2 is INSURED_EMP, and suppose that each insured employee corresponds to exactly one EMP. Then a key of EMP (e.g., SS#) can be copied into INSURED_EMP to serve as a key. For case (2), suppose that each employee has a Rank_in_Dept, and within a department, each employee has a different Rank. Then if Dept# is copied into EMP, (Rank ∪ Dept#) becomes a key.

—*Creating a surrogate key for an entity **E2**:*

Add to the entity scheme a surrogate attribute that is null-not-allowed and a key. We postulate that an arbitrary surrogate does not affect the information content.

3.6 Transforming a Relationship to an Entity

The decision of how to model a connection often needs to be changed. For example, since ER+ relationships cannot have attributes, if a designer wishes to attach an attribute (say, StartDate on a WORKS_IN relationship), the relationship must be transformed to an entity. Conversions also occur when m:n relationships need to be represented in a style subset that lacks that construct. The rearrangements in this section are therefore important modules for schema enhancements and for the tools that transform schemas from one model to another (Section 3.7). Their inverses would be important in reverse engineering.

Relationship-to-entity transformations replace a relationship by a new entity and two incident (1:1 or 1:n) relationships. We consider two cases, depending on whether the relationship is value determined. To simplify the discussion, we assume there is no set-key on the original relationship.

3.6.1 *Converting a Non-Value-Determined Relationship to an Entity*

—*Converting a non-value-determined relationship* R***12*** *(between entities **E1** and **E2**) into a new entity and two new relationships:*

Example. Suppose USED_SKILL relates EMP and SKILL entities. The relationship ⟨USED_SKILL⟩ is replaced by the entity USED_SKILL and the two new relationships ⟨R13⟩ and ⟨R23⟩.

EMP—(1,n)—⟨USED_SKILL⟩—(0,m)—SKILL is rearranged to:
EMP—(1,n)—⟨R13⟩—(1,1)—USED_SKILL—(1,1)—⟨R23⟩—(0,m)—SKILL

The steps in the transformation are:

(a) Create a new entity, bearing the name of the relationship. For clarity, we denote the new entity E3.

(b) Create new relationships R13 and R23.

(c) Fix minimum (m) and maximum (M) participations as shown below.

E1—(m1,M1)—⟨R12⟩—(m2,M2)—E2 is rearranged to:
E1—(m1,M1)—⟨R13⟩—(1,1)—E3—(1,1)—R23—(m2,M2)—E2

(d) Import keys from incident entities to enforce uniqueness of R12 instances:
— Identify primary keys K1 for E1, K2 for E2 (Section 3.5).
— Copy keys K1 from E1 and K2 from E2 into E3 (Section 3.2).
— Declare (K1 ∪ K2) a primary key of E3.

Step (d) enforces a subtle constraint implicit in the input schema, that a relationship's population is a (duplicate-free) set of entity pairs. (Support for multiset relationships would probably just confuse designers.) Noting that the uniqueness constraint is not a result of careful user consideration, a system might reasonably skip step (d) which enforces it.

3.6.2 *Converting a Value-Determined Relationship to an Entity*

—*Converting a value-determined relationship* R***12*** *into a new entity and two new relationships* R***13*** *and* R***23:***

The previous transformation created an entity instance for each related pair, e.g., (emp2, skill3). When the relationship is value determined, a different transformation seems more natural. For example, suppose CUSTOMER and SALESMAN have a value-determined relationship based on CityName. We want to create a new entity whose only attribute is CityName, and which contains the name of each city that has both a customer and salesman.[6] The steps in the transformation are:

(a) Replace R12 by an entity E3 with attribute set A3 that is a copy of A1 (the set of R12-value-determining attributes in E1). Null-not-allowed constraints are the same as in E1, except that not all A1 attributes can be null. Declare A3 to be the primary key of E3.

(b) Create R13 between E1 and E3, value-determined by matching E1[A1] and E3[A3]. Impose inclusion constraints: E1[A1] ⊃ E3[A3] and E2[A2] ⊃ E3[A3]. The participation constraints are shown below, where $m1^-$ denotes min(m1,1), $m1^+$ denotes max(m1,1), and $m2^-$ and $m2^+$ are defined similarly.

E1—(m1,M1)—⟨R12⟩—(m2,M2)—E2 is rearranged to:
E1—($m1^-$,1)—⟨R13⟩—($m2^+$,M2)—E3—($m1^+$,M1)—R23—($m2^-$,1)—E2

(c) If R12 had an inclusion constraint E2[A2] ⊃ E1[A1], impose the inclusion E3[A3] ⊃ E1[A1]. Otherwise, generate the (weaker) constraint that all nonnull values in (E1[A1] ∪ E2[A2]) appear in E3.

[6] We also considered two slightly different transformations and instance mappings for the conversion. One alternative creates an instance of the link entity for each city having either a customer or a salesman. The second creates a link instance for each city having a customer. Rather than complicate the semantics of value-determined relationships to distinguish these cases, we simply allow the user to manually edit the transform output.

(d) Create R23 between E2 and E3 analogously to the creation of R13.
(e) Infer additional constraints based on key or "mandatory" constraints on E1 and E2 (see Section 3.4).

3.7 Rearrangements to Translate between Data Model Styles

This section describes composite rearrangements that produce schemas containing constructs solely in the relational-style or network-style subsets of ER+ . These are not difficult once the basic rearrangements on which they are built are available. The restricted schemas can then be mapped directly to the respective target models. The *only* practical way to produce complex rearrangements (like these) may be by composition of smaller rearrangements. Otherwise it may be too difficult to get the constraints correct and to verify invertibility of the instance mapping.

3.7.1 *Transformation to Relational Style*

(a) Obtain keys for each entity (see Section 3.5).
(b) Ensure that all 1:n and 1:1 relationships are value determined and have no set-keys. To achieve this, copy key attributes across any offending relationship.
(c) Produce "link" entities to replace all m:n relationships that are not value determined.

We considered adding a step to check whether the resulting relations were normalized. However, in our limited use of DDEW, we never encountered a situation where the created schema was unnormalized (at least according to the dependencies known when the transformation was performed).

3.7.2 *Transformation to Network Style*

(a) Convert m:n relationships to "link" entities.
(b) Convert non-value-determined, reflexive relationships to entities.

3.7.3 *Transformation from Network to Hierarchical Style.* DDEW inferred missing parent constraints on 1:1 relationships, found a spanning forest through arbitrary relationship deletion, and determined that all relationships outside the forest should be handled by applications.

For obvious reasons, that is *not* a content-preserving transformation. The DDEW user guide advises the database designer to declare appropriate parents for 1:1 relationships, to delete relationships to get rid of multiparent entities, and to break cycles if present—*before* invoking the transformation.

3.8 Extending the Infrastructure to Accommodate Inheritance

Extensibility is a major requirement for any software engineering environment. For example, due to resource limitations and sponsor priorities, we did not include inheritance in 1983 during DDEW's design. Appropriate models and mapping algorithms for extended-ER (*EER*) models can now be found in the literature (e.g., Markowitz [1989]). In this section, we describe how DDEW's infrastructure would be exploited and extended to support such approaches.

A natural approach is to consider an IS-A link to be a relationship, with an extra annotation that it has the semantics of IS-A, and with participations as shown in the example: EMP—(1,1)—⟨ ⟩—(0,1)—PERSON. Beyond this simple embedding, extensions would be needed in three areas.

Display. The display system would need enhancement to provide an appropriate display (DDEW already supports arrows for Codasyl schemas). It would also be desirable to modify layout algorithms so generalizations would appear above specializations.

Data Model. ER+ would be extended with constructs that constrain subtypes of a given supertype. Although EER variants differ in the details, they tend to include constraint constructs such as *disjointness* (e.g., EMP and RETIREE are disjoint) and *covering* (the union of EXEMPT_EMP and NONEXEMPT_EMP includes all instances of EMP). Some constructs (m:n relationships, set-keys) would be excluded from EER-style schemas.

Transformations. New tools would be needed, in order to reverse-engineer existing schemas to EER by inferring IS-A relationships and to translate EER schemas to relational implementations. For reverse engineering, Johannesson and Kalman [1989] and Oertly and Schiller [1989] contain algorithms that infer or guess IS-A relationships. The DDEW infrastructure contributes to implementing these algorithms by supplying rigorous rearrangements for attribute removal, plus relationship inference heuristics.

When selecting a relational-style implementation of an EER schema, *any* subset of a supertype's attributes could be copied to the subtypes (and usually deleted from the supertype). An alternative physical implementation is to copy attributes from the subtypes into the supertype (with null allowed). In both cases, DDEW's attribute-copy rearrangements would do the necessary lower-level work and create the necessary constraints. Some new low-level rearrangements should also be supplied, to remove entities that are redundant once all their attributes have been copied. The choice among alternative relational implementations is difficult and is influenced by the anticipated transaction load, and sometimes by the assumptions of existing application software.

4. HEURISTIC TOOLS THAT AUGMENT SCHEMAS

Early in the design process, the schema can be expected to be sketchy, omitting many relationships, attributes, and constraints. So in initial design, adding missing information is a higher priority than maintaining content equivalence. In this section we show how heuristic tools that make likely guesses can be combined with rearrangements and user input (including confirmation of guesses) to rapidly improve an initial specification.

The initial DDEW schema may consist only of entities, obtained from imported descriptions of relations or files, or from a collection of functional dependencies. Other times the initial design may include relationships and screen layout information that needs to be preserved. Aided by heuristic tools, designers have several options for improving the design.

—*Refine the entity definitions in a schema*, splitting entities to assure that each entity is normalized, and eliminating redundant attributes and functional dependencies (Section 4.1).

—*Refine the relationships in a schema*, adding needed relationships, eliminating spurious and redundant ones, adding constraints, and improving relationship names (Section 4.2).

—*Integrate two or more schemas* (Section 4.3).

4.1 Normalization for Redesign of Existing Schemas

DDEW includes a utility that transforms a collection of FDs into a set of Third Normal Form entities. Since this normalization algorithm is routine, we will not discuss it further. Instead, we examine how normalization ideas, which were developed to produce an initial relational schema from functional dependencies, apply in *redesign* of an ER+ schema. The redesign problem is important, since evolution of existing systems consumes much of the DP budget. Normalization of ER+ schemas takes the design through six steps:

(1) an initial ER+ diagram display;
(2) set of functional dependencies;
(3) a cover of the original dependencies;
(4) keyed normalized entities;
(5) suggestions for new constructs, relative to the input schema;
(6) a new, user-approved displayable ER+ schema.

The traditional approach to normalization (comprising stages (2)–(4)) cannot provide a design tool that manipulates *arbitrary* ER+ schemas, for three reasons. First, normalization does not handle the problem of preserving ER+ information such as relationship names and diagram positions. Second, normalization theory [Cosmadakis and Kanellakis 1984] is inadequate for "bad" schemas, whose attributes may be null, or which include arbitrary patterns of functional and inclusion dependencies. When the input schema is equivalent to an "ER-compatible" schema [Markowitz and Shoshani 1989], normalization can be used, but the transform to ER compatibility alters a schema with which the user is familiar, and may increase its size. Third, the universal relation scheme assumption [Maier 1983] (denoted *URSA*) cannot be relied on. Certainly, few corporate data administrators can determine whether it holds in their databases.

Below, we discuss the pragmatics of various approaches to normalization. Section 4.1.1 compares approaches that normalize each entity separately versus approaches that deal with all entities' dependencies simultaneously. Section 4.1.2 deals with the fact that an entities-only schema is an unsatisfactory result, particularly for a user who supplied a full ER+ diagram.

4.1.1 *Local versus Global Approaches to Normalization.* We compare two approaches to using normalization starting from an existing ER+ (or relational) schema. The first approach accomplishes little, but does it rigorously. DDEW used the second approach, which makes faster progress, often in the

right direction. We relied on interactions with the user to prevent serious mistakes.

Local normalization takes the functional dependencies defined for each entity and performs a separate normalization. Dependencies may be obtained from key constraints and from 1:1 value-determined relationships [Ling 1985]. Since the algorithm works one entity at a time, it cannot detect redundancy that involves multiple entities. Each entry does indeed satisfy URSA, so normalization preserves all dependencies.

Unfortunately, we suspect that few users will have the knowledge and patience to specify the nonkey FDs needed to cause changes. Naive users do not understand such dependencies. A sophisticated user who discovers a nonkey dependency seems just as well served by a manual capability to redesign the entity as it ought to be. Therefore we decided that local normalization *for redesign* was not a priority.[7]

Global normalization throws functional dependencies from *all* entities into a "soup of attributes" and uses normalization to simplify the ensemble (thus subsuming local normalization). For the duration of normalization, attributes of the same name in different entities are considered identical (i.e., we temporarily adopt URSA). The URSA assumption is very unreliable across entities, so DDEW treats the results of entity normalization solely as *suggestions to the designer*, who decides whether to merge entities with apparently identical keys, or where incident relationships should be reconnected after an entity has been split (normalized).

Local and global normalization represent extreme views about which attributes in different entities are to be considered identical. A middle path might be to obtain additional information to drive normalization, from a dictionary of explicit synonyms specified by the user, or from constraint information (inclusion dependencies and matches between attributes in value-determined relationships).

4.1.2 *Automatic Normalization While Preserving the Input Schema.* An existing input schema has relationships, constraints, names, and diagram layout; normalization yields just entities and their functional dependencies. If DDEW were to be extended to have more automated global normalization, we would need to accommodate the entities-only results from the Normalizer, without losing all the additional information. A long-term direction would be to build a very flexible schema integrator and to have it combine the original schema with the Normalizer output, with a default to use names and display information from the original schema. Since view integration is also used for other purposes, one can justify considerable effort to implement its heuristics and user interactions.

A simpler approach would be to extend normalization to *tag* each dependency with its source entity name. An entity in the resulting entities-only

[7] *Automatically* splitting and merging entities would require substantial new code to reconnect the incident relationships, provide good names for the new entities, and generate a layout for the resulting diagram.

schema is tagged with the names associated with all dependencies used in its creation. This mechanism makes it easier to establish correspondences among entities in the two schemas (and hence to preserve each entity's name and diagram position). Going further, one could include special logic in the Normalizer to handle splitting and merging of entities. When entities are merged, their incident relationships can be redirected. When an entity is split, relationships incident to the split entity can be allocated based on the fate of attributes involved in value determination constraints.

4.2 Relationship Heuristics

Schemas need relationships—a display of 40 unconnected boxes is unusable. But schemas containing just entities do arise, from file descriptions, from a relational catalog, or as the output of the global normalization algorithm. Therefore DDEW included transformations (both heuristics and rearrangements) that inferred and refined a set of relationships. After relationship refinement, we automatically generated diagram positions for nodes and routings for connecting lines. The results could then be edited interactively.

The work described below improves on several aspects of existing technology. First, our input schema need not contain inclusion dependencies (which are unavailable in most file definitions, existing relational schemas, and schemas synthesized from sets of functional dependencies). Second, we do not restrict inclusions to single attributes or to acyclic patterns [Cosmadakis and Kanellakis 1984]. Finally, we introduce new, very general rules for identifying redundant relationships. However, unlike Casanova and Amarel de Sa [1984], our algorithm is not content-preserving, since it adds information by guessing likely relationships and inclusions. Also, unlike Johannesson and Kalman [1989], we do not identify IS-A hierarchies or map existing relations to relationships.

4.2.1 *Heuristics for Initial Synthesis of Relationships.* This section describes and illustrates six steps (a)–(f) to identify modifications to a set of relationships. At the end of the section, we discuss issues that arise if relationships are present initially. We use the following example schema, with primary keys underlined:

DEPT(Dept#,Address)	EMP(SS#,Dept#,Name,Age)
PROJECT(Proj#,Dept#,Budget)	TASK(Task#,Name,Proj#,Due_Date)

(*a*) *Create Relationships When Attributes Match.* If two entities include a single attribute of the same name, create a relationship, value-determined by that attribute. If there are several pairs of attributes with matching names, create a single relationship, value-determined by that set of attributes.

A decision to create a relationship determined by either nonkey attributes or by multiple matching attributes ought to be reviewed by the designer. With nonkey attributes, the relationship is likely to be spurious. With multiple matching attributes, some of the matches may be spurious, or they may represent *different* relationships. The heuristic can be improved incremen-

tally, by making detection of matching attributes more intelligent, e.g., by identifying matches based on knowledge of prefixes and suffixes (e.g., DEPT, Dept#, Dept_No), a thesaurus (Worker, Employee), or sound-alikes.

Example. The above schema is augmented by the following five hypothesized relationships (later steps will find one of these to be redundant and another meaningless):

Synthesized relationships	determined by
DEPT_EMP	Dept#
DEPT_PROJECT	Dept#
EMP_PROJECT	Dept# /* redundant */
PROJECT_TASK	Proj#
EMP_TASK	Name /* meaningless */

(*b*) *Infer Maximum Participations.* If a value-determined relationship includes the key attributes of one entity, then maximum participation in that direction is 1 (using the rearrangements in Section 3.4). For example, maximum participation is 1 from EMP in DEPT_EMP; from PROJECT in DEPT_PROJECT; from TASK in PROJECT_TASK.

(*c*) *Guess Inclusion Pattern from Foreign-Key References.* Suppose R is value-determined by matching E1[A1] and E2[A2], while maximum participation is 1 from E2 to E1, and unconstrained in the other direction. The structure appears to be a foreign-key reference for a 1:n relationship, so guess that E1[A1] $\supset$ E2[A2].

Example. The following inclusions are guessed:

DEPT[Dept#] $\supset$ EMP[Dept#]
DEPT[Dept#] $\supset$ PROJECT[Dept#]
PROJECT[Proj#] $\supset$ TASK[Proj#]

Example. We do not guess inclusions for 1:1 relationships. To see why, notice that in the following schema the three relationships that one can infer based on SS# give no syntactic clue for guessing inclusions (or which concept is the generalization):

PERSON(SS#,Name,Age)
EMP(SS#,Salary)
STUDENT(SS#,Grade_Point)

As can be seen from the previous examples, relationship creation—despite its benefits—suffers from redundancy, nonsense, and omissions. To ameliorate these problems, DDEW included steps (d)–(f) below.

(*d*) *Identify and Delete Redundant Relationships.* Relationships that are provably the composition of other relationships are identified and may be deleted. This step is needed because schemas synthesized by rules (a)–(c) contain some relationships that humans see as derived rather than

fundamental. Section 4.2.2 describes the detection algorithm, which provides some new theoretical results.

(*e*) *Delete Relationships Derived from Meaningless Matches.* While many relationships created by our heuristics are meaningful, some are based on nonsensical matches (e.g., TASK.Name and EMP.Name). Exhortation to check tool results will be *quite* ineffective. Therefore, DDEW has two mechanisms to bring such situations to the user's attention. First, value-determined relationships based on nonkeys are visually distinguished, since they are particularly likely to be spurious. Second, our tool draws the user's attention to each created relationship by requesting a meaningful name to replace the temporary name.[8] Manual deletion is fast for relationships deemed spurious by the user.

(*f*) *Detect Inadvertent Omissions* (*where possible*). DDEW includes a primitive navigational, model-independent language in which designers are asked to specify transactions, as part of conceptual design. These transactions supply an additional check on the completeness of a schema. If a designer-specified transaction references two entities that are not connected by a relationship, DDEW generates a warning. After this step, redundant attributes may be removed (Section 3.3).

For schemas that already have relationships, steps (a)–(f) can be applied to suggest additions, deletions, and new constraints. For example, if these steps were applied to a schema produced by view integration, they could identify relationships between entities that came from different user views and identify redundant relationships introduced by view integration. As with normalization, it seems unwise to simply modify a schema based on such suggestions. It is preferable to treat the suggested relationships as part of a new schema that is to be integrated with the original one, which is generally more trustworthy.

4.2.2 *Redundant Relationships.* Step (a) above may also create unnecessary relationships. Unnecessary relationships clutter a schema, making it hard to understand, transform, and lay out clearly in a diagram. Additionally, value-determination constraints on an unnecessary relationship can prevent removal of a redundant attribute along another relationship. This section therefore explores criteria for identifying relationships that can be deleted. The difficulty lies first in getting a suitable definition of "unnecessary," and second, in finding an easy way to determine which relationships satisfy that definition.

Our result is a composition-based theory that appears to be substantially superior to competing approaches in its ability to eliminate unnecessary relationships and in its minimal need for human-supplied information. A relationship R is considered to be *unnecessary* if it is both *semantics-unneces-*

[8]The relationship creation tool generates a temporary name as the concatenation of entity names, with an underscore as separator. Duplicate names are detected and given an integer suffix.

sary and *content-unnecessary*. Unnecessary relationships are deleted at step (d) of 4.2.1.

—R is *semantics-unnecessary* if (1) it is the composition of other relationships in the schema (so that its population can be obtained by joining together those other relationships) and (2) the user has given no hint that the relationship is semantically useful (i.e., neither created it nor renamed it manually).

—R is *content-unnecessary* if deleting R leaves the schema's information content unaltered. A value-determined relationship's population is deducible from entity populations. Hence it will be content-unnecessary if all attached constraints are redundant. Non-value-determined relationships are content-unnecessary only in the rare cases where someone has supplied an explicit formula for computing the relationship's population (Section 4.2.3).

The concept of "semantics-unnecessary" is an attempt to capture those relationships the designer really wants to keep visible. Information content is not a sufficient criterion—for example, it would lead to the deletion of nearly all relationships as soon as they had been converted to be value-determined. This deletion would lose the structure of the schema and the opportunity to attach additional constraints.

DDEW uses the Composition Theorem below to test the semantics-unnecessary condition. The main usage was for relationships created by step 4.2.1(a); these relationships were value-determined, had few known constraints, and often had not yet been seen by the user (since (a)–(c) were automated).

COMPOSITION THEOREM FOR RELATIONSHIPS THAT SHARE COMMON ATTRIBUTES. *Consider the value-determined relationships shown in Figure* 4, *in which* R13 *and* R23 *are determined by a subset of the attributes determining* R12. (*The* A *and* B *attributes match, and* A′ *and* B′ *match.*) *Suppose also that* E2[B,B′] ⊃ E1[A,A′]. *Then* R13 *is the composition of* R12 *and* R23.

PROOF. To show that R13 equals the composition R12*R23, we show set containment in each direction.

(1) To prove R12*R23 *is contained in* R13, suppose (e1,e3) ∈ R12*R23. By definition of composition, R12*R23 = {(e1,e3) | ∃e2 such that (e1,e2) ∈ R12 and (e2,e3) ∈ R23}. Because R12 is value determined, e1[A,A′] = e2[B,B′]. Hence, e1[A] = e2[B]. Since R23 is value determined, e2[B] = e3[C]. Hence e1[A] = e3[C], so (e1,e3) ∈ R13.

(2) To prove R12*R23 *contains* R13, suppose (e1,e3) ∈ R13. Then e1[A] = e3[C], and e1[A] is nonnull. Since E2[B,B′] ⊃ E1[A,A′], ∃e2 such that e2[B,B′] = e1[A,A′]. Hence e2[B] = e1[A], so e2[B] = e3[C]. The equalities and the fact that R12 and R23 are value determined imply that (e1,e2) ∈ R12 and (e2,e3) ∈ R23. Hence (e1,e3) is in R12*R23. □

Example: "*Horizontal*" *Shortcut*. In Figure 5, the relationship EMP_PROJECT guessed in 4.2.1(a) is redundant. EMP_PROJECT must be the

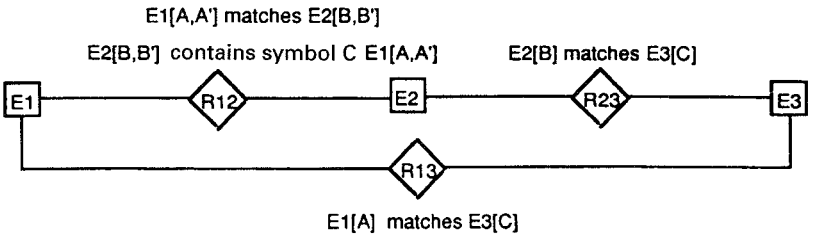


Fig. 4. Composition of relationships.

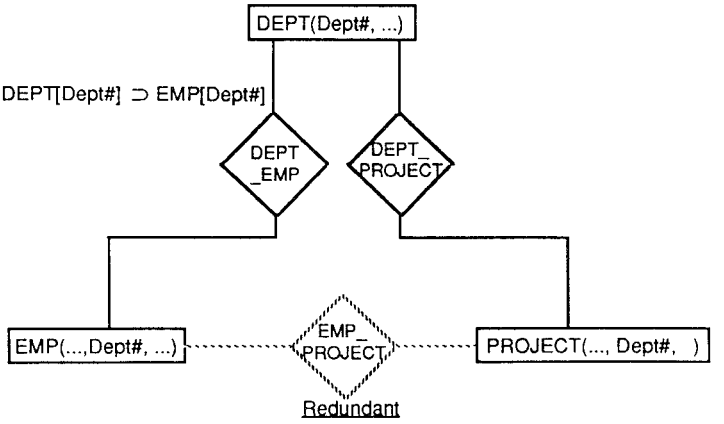
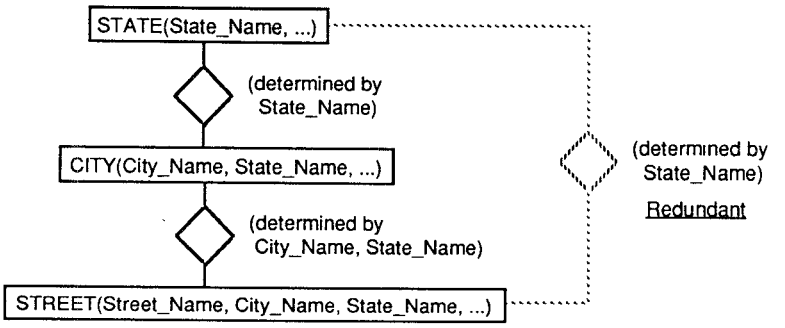


Fig. 5. "Horizontal" shortcut.

composition of DEPT_PROJECT and DEPT_EMP (by the Composition Theorem, with A = B = C = (Dept#), A′ = B′ = ϕ).

Such "horizontal-shortcut" relationships can *substantially* complicate a schema and indeed were the spur for creating our theory of relationship deletion. If n entity types reference DEPT using Dept# as a foreign key, then $n(n-1)/2$ redundant "horizontal" relationships are generated. As long as one inclusion dependency was available, our algorithm was able to delete these relationships.

Example: *"Vertical" Shortcut.* In Figure 6, the relationship between STATE and STREET is redundant. The Composition Theorem appears to be far more powerful than competing techniques (discussed in the next section). It requires very little information from users and applies in a wide range of situations. It is oblivious to keys and requires only a single inclusion to be present. Unnecessary relationships created in 4.2.1(a) often are deleted without ever being seen by the designer—a major advantage. To see the generality of situations that the Composition Theorem can detect, note that the vertical shortcut applies (the redundant relationship can be deleted) even though the database can contain (rural) streets with null city and/or state



Assume CITY[City_Name, State_Name] ⊃ STREET[City_Name, State_Name].

Fig. 6. "Vertical" shortcut.

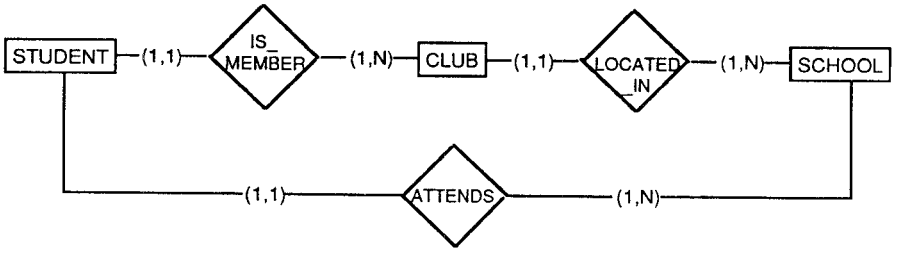


Fig. 7. Is relationship "ATTENDS" redundant?

names, (foreign) cities that are not in states in the database, states whose cities are not part of the database, and identically named streets within a city.

4.2.3 *Related Work on Vertical-Shortcut Rules.* Alternative approaches to identifying unnecessary relationships appear to be significantly less powerful than ours. We consider two cases: (1) vertical-shortcut rules without value determination and (2) approaches based on inclusion dependencies.

Vertical-shortcut rules have been used in design algorithms proposed previously [Klug 1979; Teorey et al. 1986]. But without value determination, there is no formal information in the schema to enable a tool to determine whether a relationship is redundant. Consider the example in Figure 7, in which it is recommended that the relationship ATTENDS be dropped [Teorey et al. 1986].

Segev [1987] observes that the above diagram does not itself determine whether ATTENDS is redundant. He therefore asks the user to provide additional constraints, e.g.,

IS_MEMBER(Student, Club) and ATTENDS(Student, School) ⇒
LOCATED_IN(Club, School)

We fear that there is little hope of motivating designers to supply constraints on triples of relationships—the number of triples is quite large, and straightforward manual deletion might be preferred. In ER+ , the designer's duty is composed of smaller steps, simply checking on the determining attributes of each relationship individually. Information implicit in having value-determined relationships over the same attributes then disambiguates the situation. The designer would model the above situation as follows.

Entity

STUDENT(Stud#,Club#,School#,...)
CLUB(Club#, School#,...)
SCHOOL(School#,...)

Relationship	Value-determined by
LOCATED_IN(CLUB,SCHOOL)	School#
ATTENDS(STUDENT,SCHOOL)	School#
IS_MEMBER(STUDENT,CLUB)	Club#,School#

Constraint

CLUB[Club#,School#] ⊃ STUDENT[Club#,School#]

Both the value-determined constraint is IS_MEMBER and the inclusion constraint involve multiple attributes, so it makes sense to flag them for designer attention. This notification would explicitly raise the issue of whether the attribute STUDENT[School#] that determines ATTENDS is the same as the School# at which his or her CLUB is located. In this example shown, the affirmative answer is captured by the inclusion. Thus the Composition Theorem applies, and the relationship is redundant.

A second approach is to base redundancy detection on inclusion dependencies in relational databases, under the (limiting) assumption that the only relationships to be considered are ones that involve inclusions. The theory of inference for inclusions is adequate to handle simple patterns in common cases of vertical shortcuts [Cosmadakis and Kannellakis 1984]. However, that theory is of no help if one has relationships that are not based on inclusions (e.g., horizontal shortcuts) or that involve entities and relationships for which the user has not yet specified all constraints. The Composition Theorem handles all these cases, and many more besides.

4.3 View Integration

The research literature on view integration (see Batini et al. [1986] for an overview) consists principally of careful case-by-case heuristics for resolving particular types of mismatches between schemas (e.g., conditionally mergeable relationships [Navathe et al. 1986]). A contrasting approach is given in Biskup and Convent [1986] and Casanova and Vidal [1983], which give a rigorous theoretical treatment of integration of relational views.

As system builders, our goals were more global and pragmatic. Over schemas containing numerous different constructs (entities, relationships, constraints, datatype declarations, annotations), we needed to detect correspondences and conflicts, to acquaint the designer with the conflicts and

potential conflict resolutions and to construct automatically a global schema that preserves and combines the available information.

View integration performs integration of objects at multiple conceptual levels—schemas having entities and relationships, entities having attributes and constraints, attributes having constraints and datatypes. With minor exceptions, we used the same series of processing steps at all levels. This regularity greatly simplified implementation and user understanding. Early users of the system reacted quite positively. Although the theory and capabilities of our view integrator were limited, the contribution was that the *regularity* of view integration (based on the nesting of the information being integrated) enabled a very useful heuristic tool to be built with surprisingly little effort.

The tool helps to merge two views into a single ER schema. First, the view analyzer component of the tool detects conflicts in the two views and reports on them to the designer. Second, the designer manually resolves some or all of the conflicts in the source schemas. Third, the view synthesizer component of the tool merges them into one, resolving any remaining conflicts. (We give more details on these three steps next.) The entire view integration process is tracked graphically at the schema level in DDEW's design tree.

Conflict Detection by the View Analyzer. The first step is to run the *view analyzer* that detects synonym and homonym conflicts, at several levels of detail. This tool examines names, key patterns, and other structural similarities. For example, suppose E1 and E1′ in Schema 1 correspond to E2 and E2′ in Schema 2. If Schema 1 has a relationship between E1 and E1′, and Schema 2 has a relationship between E2 and E2′, the analyzer guesses that the relationships correspond, and informs the user. For objects that appear to correspond, the analyzer detects conflicts in their constraints and other details. For example, corresponding entities may not have identical key declarations and attributes; corresponding relationships may differ in their participation constraints, inclusions, and value determination. To aid ease-of-use, DDEW avoided technical terminology when reporting results to the designer. Rather than referring to "synonyms," the view analysis report flagged objects with "different types but the same names" in both schemas. Entity and relationship conflicts were flagged as the most serious; conflicts involving attributes were described as less of a problem.

Conflict Resolution. Next, in a manual phase, the designer is asked to modify one or both schemas to resolve ambiguities and conflicts. (This corresponds to providing integration information in Navathe et al. [1986].) While examining the tool report from the conflict detection step, the designer renames, modifies, or deletes entities, relationships, and attributes as needed. Objects which are not the same should be given different names. Conflicts at the attribute level are not so serious, but differences in relationship participation may signal totally different ways of thinking about the database. For example, in one schema, employees *must* belong to a department; in the other, EMP entities need not be linked to DEPTs. The designer must decide

which semantics are correct and adjust the schemas accordingly. Another strategy is to recognize when objects are part of a generalization hierarchy.

View Synthesis. Finally, the automated *view synthesizer* tool merges objects with the same names, comparing and matching attribute lists and constraints. For each type of construct (e.g., entities, attributes, datatypes), there is a rule for combining information (union, intersection, or designation of one schema as a more reliable "master" schema). The tool also adds any additional (nonmatching) objects from the two schemas. Unfortunately, it did not produce instance mappings among the various schemas.

5. THE ROLE OF DATABASE THEORY IN PRACTICAL DATABASE DESIGN

Theory affects database design practice mostly by being embodied in design systems. Consequently, the applicability of dependency theory and other formal models to database design must be judged in terms of how the theory can contribute to such systems. During initial system design (1983), the only ER schema rearrangements we found in the literature were: elimination of redundant constraints and schema normalization based on functional dependencies [D'Atri and Sacca 1984]. But this latter technique required a dubious assumption (URSA), forbade nulls and other constraints, and was not in a model that a design system could present to its users. Techniques for rigorous transformations between models (e.g., Casanova and Amarel de Sa [1984] and Markowitz and Shoshani [1989]) provided a good theoretical setting, but only for a relational target DBMS.

Some recent work has gone further, for example, using a unified underlying model to support heterogeneous data model mappings [Kalinichenko 1990], or offering the designer a choice of alternative logical transformations on a conceptual generalization hierarchy [Oertly and Schiller 1989]. There have also been improvements to underlying algorithms such as normalization [Diederich and Milton 1989], to constraint specification [Markowitz 1990], and to graphical formulation of design semantics and process [Sockut and Malhotra 1988].

In fairness to theoreticians, while system builders have sometimes complained that much database theory is irrelevant to database design, they have published few explicit suggestions about what sorts of results are needed. Next, we contrast the interests of the two groups and then suggest practically motivated theoretical questions that can be asked about a new concept.

For an overview of current automated design tools, both research-oriented and commercial, see Ram [1993] and Reiner [1991].

5.1 Differences in Perspective between Theoreticians and System Builders

Differing views about what is critical in a design system may be one cause of the mismatch between database theory and the needs of system builders. To the theoretical community, the crucial issues are well-specified, elegant, formal models, possibly incorporated into tools that are in some sense correct and complete. However, in current commercial database design systems, the bedrock functionality is to capture, store, and display information, i.e., to

support an intelligent wall chart. Theoretical completeness of the set of transformations is not a burning issue, since a low-level editor has the power to make any change. Global consistency is *not* assured after each edit operation—it is accepted that parts of a design (e.g., requirements and the conceptual schema) are sometimes out of synch.

Ideally, a practical design system will capture, manipulate, and propagate any kind of information it is handed. It must therefore have an information model that includes not only the formal aspects of the data model, but also associated information that receives little theoretical attention, such as an entity's name, screen position, creation date, person responsible, arbitrary constraint predicates, and free-text commentary. Transformation and inter-model translation tools cannot manipulate just the formal schema; a designer would hate to start with a fully annotated ER diagram and end up with a set of normalized relations! Though some formal tools exist, they are decidedly secondary in the real world.

System builders are driven by the fact that all steps in the design process *must* be accomplished, so that any action that a tool does not perform must be handled by the designer. *Imperfect* tools plus human guidance and oversight may lead to better designs than an unaided designer with a huge clerical burden and no help in avoiding careless errors. Modules of functionality are therefore important; doing *part of* a job is still helpful.

Even when a transformation removes a theoretical difficulty, it may impose an unreasonable practical cost. For example, entity splitting (based on different roles) removes cycles from schemas, but increases the size of the schema that a designer must comprehend. Similarly, a schema with null-allowed attributes may be transformed to one without nulls. But there may be excessive costs in forcing the designer to look at an unfamiliar or larger schema, to rewrite code, or to endure bad performance.

5.2 Suggested Additions to the Theoretical Agenda

When attempting to apply existing theory during the design of DDEW, we encountered several kinds of issues that required theoretical expertise but were generally omitted from theoretical papers. Theory can contribute more to the design of real systems if these concerns—comprehensibility, robustness, and graceful degradation—receive higher priority in research efforts and also in surveys of available theory.

Comprehensibility. Even the best algorithms are useless to DDEW, or any other design system, if input data cannot be obtained from the designer. The formulations in the mathematical literature naturally emphasize generality, elegance, and mathematical convenience. But system builders need *comprehensible* displays or dialogs that can elicit dependencies and other information from a corporate data administrator. For complex dependencies, theoreticians are better qualified than system builders to suggest accessible alternative formulations equivalent to the original. Where there is no accessible formulation—even a slightly weakened one—the dependency will surely not be directly usable in DDEW or any other system. This may be a signal (ideally, detected early) that the dependency is not fundamental.

Robustness. Both an initial list of dependencies and a first-cut database design gathered from a designer ought to be reasonably robust against likely schema changes. Unfortunately, during the design of DDEW we did not find any analyses of robustness for the more complex types of dependency. After the system was completed, Ling [1986] proved that existing multivalued dependencies (MVDs) must be reexamined after addition of any attributes. Since attribute addition is common even in mature schemas, a wise administrator may decide that other activities will yield a greater return in schema quality. This weakness alone seems sufficient to rule out the use of MVDs in practical database design.

Graceful Degradation of Tool Algorithms. Many algorithms impose substantial restrictions on the input schema (e.g., typed inclusions, single-attribute keys, no nulls). But the needs of a single algorithm or tool cannot determine what information should be captured by the system. Furthermore, a tool need not be considered inapplicable just because a schema contains some local problems. Two ways that a tool could be modified to degrade gracefully are outlined next.

First, offending portions of a schema can sometimes be left untransformed. Suppose that an object (entity, relationship, or attribute) is subject to a constraint that is not understood by some tool. The tool can then treat this constraint as uninterpreted and refrain from modifying that object in producing the result schema. The damage is localized. Additionally, this technique makes the system must more extensible—it is not necessary for the internal code of each tool to know all of the types of information that can be captured by the system. New constraints can be added without modifying the implementation of existing tools. This "no-op" strategy does require that the transformation's input and output be represented in the same data model. Superficially, this approach bears some relation to the encapsulation encountered in object-oriented development environments; the difference is that here the hidden information is *essential*, but is just not *manipulable in detail* by older tool versions.

A second approach is for the tool to produce its output as if the difficulty did not exist, but to identify part of its output schema as unreliable. This can reduce the designer's clerical burden, while leaving little danger of an unexamined error. Displaying dubious decisions or missing information in red is far more effective than urging designers to examine the entire output schema.

Partially Specified Schemas. Existing theories of schema equivalence, including ours, assume that one manipulates database schemes. Our tools were also able to transform *partially specified schemas*, which had "don't know" (denoted "?") entries for participations and some other constraints. It was clearly wrong to treat "?" as "no constraint," so we attempted to propagate the uncertainty appropriately. We are currently producing a theory to guide "correct" transformations of schemas that include "?".

Surveys Aimed Also at Practitioners. System builders work under severe time pressure. At best, some may read survey papers and follow promising

references. Therefore published surveys need to contain pointers on practically motivated questions such as the ones above.

6. CONCLUSIONS

We described the tools in DDEW, a comprehensive system for database design, and showed how they work together to support the design process. The system uses a rigorous, information-content-preserving approach to schema transformation, but combines it with heuristics, guesswork, and user interactions. Within the integrating framework of DDEW, we took unusual approaches in three areas:

—A unified underlying data model (ER+) for all processing, including conceptual, logical, and physical design.
—A reusable and composable library of content-preserving rearrangement transformations of varying granularity.
—Heuristic tools for normalization, relationship refinement, and view integration that improve schemas in a nonrigorous but interactive fashion.

Unified Underlying Data Model

Building the system over a unified underlying data model enabled the same tool code to be run under many different circumstances, minimizing both the learning burden on users and the implementation effort by system builders. Both the code and the theory on which it is based are effectively shared among multiple target models, including the three classical data models. The contribution of ER+ is less in its specific constructs than in showing that it is possible to meet a very large number of system needs, while keeping model complexity and redundancy manageable. Because ER+ is ubiquitous, DDEW is not limited by a deep, permanent decision about *which* data is visible at each step of design. Tools are also robust; if an intermodel translation tool cannot handle part of a schema, that part is left unchanged; the schema remains a valid ER+ schema.

Content-Preserving Rearrangements

A design system ought to ensure that a transformation between equivalent schemas will not introduce new errors into the modeling of the real world. If late transformations can introduce errors, correctness depends on the designer's final check, instead of on the union of all accuracy checks in the design process. DDEW showed the feasibility, utility, and costs of a more rigorous approach—defining and exploiting *rearrangement* transformations whose outputs are guaranteed to be content equivalent to their inputs.

We presented a core set of small but fundamental rearrangements, from which we were able to build larger tools and transformations that were still formally justified. Rearrangements in DDEW included: replicating the attributes of an entity in related entities and (inversely) eliminating such replication; converting a complicated relationship to an entity and two

simpler relationships; inferring additional constraints and (inversely) removing redundant constraints; creating keys; and translating among data models.

Heuristic Tools

We described heuristic tools that attempt to improve a schema, often by adding missing information. In these tools, unreliable techniques are bolstered by system-guided user interactions to remove errors. For *normalization*, we showed that a user/tool partnership allows use of a less reliable but more effective "global" normalization algorithm. To *refine the set of relationships*, we alternated heuristic steps and rigorous inferences. In our experience, it was possible to synthesize automatically a credible set of relationships for a relational schema. Declarations of value-determined relationships provided crucial information for recognizing *redundancy*. Lastly, we provided *view integration* for a complex system supporting many target data models, using a consistent multilevel approach to detect and resolve conflicts.

The Role of Database Theory

We examined the relevance of database theory to building these practically motivated tools and contrasted the paradigms of system builders with those of theoreticians. Formal considerations played an important role in our design, but the research literature was of surprisingly little use (except on the subjects of schema equivalence and normalization of functional dependencies). We needed to develop our own formulations to handle general sets of constraints and multiple data models with one body of software. Despite assertions that sophisticated dependency theory can aid database design, the theory generally imposed unacceptable comprehension and schema analysis burdens on designers, and techniques were insufficiently robust. This was partially responsible for our emphasis on heuristics and user interaction.

Finally, we suggested an agenda for making theoretical work more useful to system builders. Theoretical algorithms should be concerned with *all* the information associated with a diagram and should exploit interaction with the designer. Design information should be reasonably robust against later schema changes and must be derivable from declarations that corporate data administrators can supply. Tool algorithms should expect assumptions to be violated and degrade gracefully. Theory should help a system support design in a user's favorite data model.

Summary

We believe that a combination of heuristics, rigorous transformations, and planned interactions with the designer can indeed lead to powerful database design tools. For initial conceptual design, where the input information is unreliable, DDEW emphasizes heuristic tools and human interaction with tool results. Once the schema becomes an accurate reflection of the real world, further transformations are formally justified and preserve the information content of the schema. Throughout, the unified underlying data

model reduces redundancy and aids both heuristic and content-preserving transformations.

APPENDIX: PROOFS FOR SECTION 3

This Appendix contains proofs that the formal transformations in Section 3 are indeed rearrangements. Below, E1 and E2 denote entity types, and R denotes a relationship between them. In the proofs, S1 and S2 denote the input and output schemas of the transformation in question. s1 (and s2) denote schema instantiations conforming to the structure and constraints of S1 (respectively, S2). Each proof is preceded by a statement of the transformation, identical to the statement in Section 3.

A direct proof that a transformation is a rearrangement must demonstrate that there exists an instance mapping (mapping of instances of source and result schemas) that is injective, surjective, total, and generic. All reasonable mappings are "natural," so that condition will not be discussed further.

The proofs were simplified by using the following general techniques:

(1) *Composition of rearrangements*: A large rearrangement can sometimes be designed as a sequence of small steps that are already known to be rearrangements. Since the composition of injective functions is injective, of total functions is total, and of surjective functions is surjective, the composition of rearrangements is a rearrangement.

(2) *Constraint rearrangements*: When a transformation changes only constraints, then the "instance mapping" in the proof is just the identity mapping. Proof complexity is reduced because the instance mapping needs no complicated definition and is injective (1-1). One need only verify surjective and total, i.e., that exactly the same set of schemas satisfy the original and modified sets of constraints.

(3) *Perturbing a known rearrangement*: Consider a situation where a new transformation that is alleged to be a rearrangement handles a tricky constraint and where one already knows of a rearrangement that handles schemas without that constraint. The verification of the difficult case can begin by using the same instance mapping I as the easy case. One need only verify total and surjective, since the instance mapping is known to be injective. Hence, it is sufficient to define a mapping I^{-1} such that for any schema s2 satisfying S2, $I^{-1}(s2)$ satisfies S1 and $I(I^{-1}(s2)) = s2$. Some combination of these techniques may help in transporting theory among data models.

Key Copying

—*Copying a key* A1 *of* E1 *across a non-value-determined relationship* R *from* E1 *to* E2:

Preconditions for Applicability. R has at most one E1 for each E2, and is non-value-determined. A1 contains a key of E1 (i.e., is a key or a superset of a key). E1[A1] is null-not-allowed.

Result. Attributes A1 are added to E2. R has the constraint that it is value-determined by A1. E1[A1] ⊃ E2[A2]. If R was mandatory from E2, then E2[A1] is null-not-allowed.

The instance mapping I leaves all entities and relationships unchanged, except that each instance e2 of E2 is extended with attributes A1 representing a copy of the related E1[A1] (which was assumed to be unique). If there is no related E1, E2[A1] is null for each attribute in A1. The inverse map I^{-1} is simply to delete A1 attributes from e2.

Total. Old constraints are unaffected. e1[A1] and e2[A1] will be equal and the inclusion constraint satisfied if e1 and e2 are R-related, since E2[A1] came from E1[A1]. And since E1[A1] is a key, e2[A1] will not match any e1′ distinct from e1; hence the other half of the value determination condition is satisfied.

Surjective. I^{-1}(s2) is identical to s2 except that attributes of e2[A1] are deleted. No constraint of S1 mentions e2[A1], and S has fewer constraints than S2, so no new constraint violations could have occurred.

Injective. Suppose distinct instantiations s1 and s1′ are mapped to the same s2. The only object where they can differ is E2, since other instantiations are unchanged by I. Let e2 be in population(E2 ∈ s1) but not in population(E2 ∈ s1′) (that is, in population(E2) in instantiation s1 or s2). Then I(E2 ∈ s1) includes an instance extending e2 with e1[A1], while I(E2 ∈ s1′) does not—a contradiction.

Additional Attribute Copying

—*Copying attributes* A3 *across a value-determined relationship* R *from* E1 *to* E2:

Preconditions for Applicability. R has at most one E1 for each E2 and is value-determined with E1[A1] matching E2[A2]. A1 contains a key of E1.

Result. E2 has the additional attributes A3. E1[A1 ∪ A3] ⊃ E2[A2 ∪ A3].

The instantiation mapping is the same as for key copying.

Injective. The key-copying proof of injective still applies.

Total. The only new constraint is inclusion; it is obviously satisfied.

Surjective. The only constraint of S1 that does not appear in S2 is the inclusion E1[A1] ⊃ E2[A2], which is strictly weaker than E1[A1 ∪ A3] ⊃ E2[A2 ∪ A3].

Attribute removal is the inverse of attribute copying, and hence is a rearrangement.

Inferring Constraints for Value-Determined Relationships

—*Inferring maximum participation*:

If A1 contains a key of E1, impose the (maximum participation) constraint that at most one E1 instance participates in R12 for each E2.

PROOF. Suppose e1 and e1′ are both related to e2. Then e1[A1] = e2[A2] = e1′[A1]. But since A1 contains a key, e1 = e1′. □

—*Inferring inclusions and null-not-allowed*:

If membership in R12 is *mandatory* for E2 instances, then: E1[A1] ⊃ E2[A2], and E2[A2] is null-not-allowed.

PROOF. For any e2, the required e1 that matches e2 must have the same (nonnull) value e1[A] = e2[A2]. For the next example, suppose that instead of specifying minimum participation nonzero, the user had specified null-not-allowed and inclusion constraints. □

—*Inferring minimum participations*:

If E1[A1] includes E2[A2] and E2[A2] is null-not-allowed, then minimum participation of E2 in R12 is at least 1.

PROOF. Consider any instance e2. e2[A2] must be nonnull, and hence there must be a corresponding e1 such that e1[A1] = e2[A2]. And e1 is R12-related to e2, by value determination. Hence the participation of e2 in R12 is at least one. □

Key Creation

—*Creating a primary key for a keyless entity* E2:

Suppose that E2 has minimum_participation = maximum_participation = 1 in R12. And suppose E1 has a primary key K1 that can be copied across relationship R12 to E2 (using the *key-copying* rearrangement of Section 3.2).

```
{If E1 also has maximum_participation = 1 in R12, then
    (1) Copy K1 into E2 and declare copied attributes K2 a key of E2;
else If R12 has a null-not-allowed set-key SK, then
    (2) Copy key K1 of E1 into E2 (as K2) and declare (SK ∪ K2) a key of E2;
else
    (3) create a surrogate key for E2 as described below}
```

Step (2) uses key copying, which is already known to be a rearrangement. The proofs for (2) and (3) are simple demonstrations that the extra key constraints introduced by the second and third clauses were already implied by the other constraints. (The text has already argued that surrogate creation did not really add information to the schema.)

PROOF OF (1). Suppose E1 has maximum_participation = 1 in R12. To show that K2 is a key, we show that any E2 instances e2 and e2′ such that e2[K2] = e2′[K2] must denote the same entity. Since max participation is 1, e2 and e2′ must have received their keys from different E1 instances, denoted e1 and e1′. By value determination, e1[K1] = e2[K2] = e2′[K2] = e1′[K1]. But since K1 is a key of E1, e1 = e1′; hence e2 and e2′ must be identical, i.e., K2 is indeed a key of E2.

PROOF OF (2). Suppose R12 has a null-not-allowed set key SK. Consider any pair of E2 instances such that e2[SK ∪ K2] = e2′[SK ∪ K2]. To show

that SK ∪ K2 is a key, we show that e2 and e2′ must denote the same entity. By the same argument as above, e2 and e2′ must correspond to the same entity e1. So we know e2 and e2′ agree on SK and are related to the same instance of E1. By definition of "set key," e2 and e2′ must be the same entity. □

Converting a Non-Value-Determined Relationship to Entity

For convenience in proofs, we define a new model construct, a *multiset relationship*, whose population is a multiset of entity pairs. A relationship is a multiset relationship subject to the constraint that the entity pairs not be duplicates. Our proof strategy is to show first that steps (a)–(c) of the transformation in Section 3.6.1 constitute a rearrangement for a multiset relationship R.12, and then to bring in the uniqueness constraint.

Multiset, Non-Value-Determined Relationship to Entity

—*Converting a non-value-determined, multiset relationship* R12 (*between entities* E1 *and* E2) *into a new entity and two new relationships*:

The steps in the transformation are:

(a) Create a new entity, bearing the name of the relationship. Here we denote the new entity E3.
(b) Create new relationships R13 and R23, constrained to be sets rather than multisets.
(c) Fix minimum (m) and maximum (M) participations as shown below.

E1—(m1,M1)—⟨R12⟩—(m2,M2)—E2 is rearranged to:
E1—(m1,M1)—⟨R13⟩—(1,1)—E3—(1,1)—R23—(m2,M2)—E2

PROOF. Define an instance map I that leaves objects other than R12 unchanged. It populates E2, R13, and R23 by:

population(E3) = one instance for each R12 instance (e1,e2).
population(R13) = {(e1,e3) | e3 corresponds to an R12 instance (e1,x) for some x}
population(R23) = {(e2,e3) | e3 corresponds to an R12 instance (e2,x) for some x}

Total. E3 has no constraints, so the only constraints to verify are the participation constraints imposed on R13 and R23. These follow straightforwardly from the constraints on R12.

Surjective. Given an instantiation s2, obtain I^{-1}(s2) by deleting R13, R23, and E3 and defining population(R12) as the multiset containing an instance (e1,e2) of R12 for each (e1,e3,e2) where (e1,e3) was in R13 and (e2,e3) was in R23. We first verify the new constraints, i.e., participation constraints on R12. Consider the constraint that each e1 have minimum participation k, i.e., at least k pairs of the form (e1,x) in R12. e1 had the same participation constraint in R13, so there are at least k instances of the form (e1, e3) in R13. Each e3 has exactly one R23-related e2, so there are at least k triples

(e1,e3,e2) where (e1,e3) is in R13 and (e2,e3) is in R23. These triples correspond by definition to pairs (e1,e2) in R12. (The proof for maximum participation is analogous.)

To verify that I does map I^{-1}(s2) to s2, we need to be more careful about handling of the relation as a multiset of entity pairs. We assume that each element of the multiset R12 has a unique label, i.e., that such multiset members can be written (L:e1,e2). Assume that when I and I^{-1} map R12 relationship instances to and from E3 entity instances, the labels are attached to the created instances. Then for each e3 instance (labeled L) in s2, I^{-1}(s2) creates an R12 instance in s1 with that same label, connecting the unique instances e1 and e2 that were related to e3. Applying I, we get back the same e3 instance.

Injective. The proof resembles that of key copying. Suppose distinct instantiations s1 and s1′ are mapped to the same s2. The only object where they can differ is R12, since other objects instantiations are unchanged by I. Hence there must be a pair (e1,e2) in one population (say, population(R12 in s1)) that is not in the other population(R12 in s1′). Then in I(s1) (but not I(s1′)) there is an instance e3 connected to e1 and e2. Hence I(s1) and I(s2) differ, a contradiction.

Modification to Deal with a Duplicate-Free Relationship

—*Converting a non-value-determined relationship* R12 (*between entities* E1 *and* E2) *into a new entity and two new relationships.*

We compose several other rearrangements to handle the constraint that relationships have no duplicate tuples. The steps are:

Identify primary keys K1 for E1, K2 for E2.

Copy keys K1 from E1 and K2 from E2 into E3.

Declare (K1 ∪ K2) a primary key of E3.

PROOF. Let S1 denote the original schema, and let $S1^-$ denote that schema with the set relationship constraint omitted. Let S2 denote the result schema and $S2^-$ denote that schema with the key constraint on E3 omitted. Let I and I^{-1} be the mappings from the proof of the previous transformation, mapping instantiations of $S1^-$ to instantiations of $S2^-$.

If the troublesome set relationship constraint did not exist, we know that I would provide the desired mapping. We will show that the restriction of I to S1 is again injective, surjective, and total.

Injective. A restriction of an injective mapping is still injective.

Total. We know that I provides a total mapping from $S1^-$ to $S2^-$. To show that it is total from S1 to S2, we show that if s1 is an instance of S1 (i.e., the set relationship constraint holds), the key constraint holds in I(s1).

The primary key attributes cannot be null, since they were copied from a primary key, over a mandatory relationship. And suppose there were two entities e3 and e3′ with the same values for (K1 ∪ K2). These attribute

values could only have come from a unique e1 (with that value of e1[K1]) and a unique e2. That is, R12 must have had two appearances of the pair (e1,e2), violating the set relationship constraint, a contradiction.

Surjective. From the previous proof, we know that I provides a surjective mapping from $S1^-$ to $S2^-$, i.e., that if s2 is an instance of $S2^-$ then I^{-1} satisfies the constraints of $S1^-$, and $I(I^{-1}(s2)) = s2$. Furthermore, we know that $I(I^{-1}(s2)) = s2$. So it remains only to show that I^{-1} maps S2 into S1, i.e., that the set relationship constraint holds in $I^{-1}(s2)$.

I^{-1} populates R12 by creating one instance for each triple (e1,e3,e2). For the set relationship constraint on R12 to be violated, S2 must contain e3 and e3′ satisfying the following: [(e1,e3) ∈ R13 and (e2,e3) ∈ R23] and [(e1,e3′) ∈ R13 and (e2,e3′) ∈ R23].

Let K1 and K2 denote the key attributes of E1 and E2 and also those same attributes within E3. Since key copying produces a value-determined relationship, we have e3[K1] = e1[K1] = e3′[K1] and e3[K2] = e2[K2] = e3′[K2]. Hence, e3[K1,K2] = E3′[K1,K2], which contradicts the constraint in S2 that [K1,K2] is a primary key. □

Converting a Value-Determined Relationship to an Entity

—*Converting a value-determined relationship* R12 (*between entities* E1 *and* E2, *by matching* E1[A1] *and* E2[A2]) *into a new entity and two new relationships* R13 *and* R23:

The steps in the transformation are:

(a) Replace R12 by an entity E3 with attribute set A3 that is a copy of A1. Null-not-allowed constraints are the same as in E1, except that not all A1 attributes can be null. Declare A3 to be the primary key of E3.

(b) Create R13 between E1 and E3, value-determined by matching E1[A1] and E3[A3]. Impose inclusion constraints: E1[A1] ⊃ E3[A3] and E2[A2] ⊃ E3[A3]. The participation constraints are shown below, where $m1^-$ denotes min(m1,1), $m1^+$ denotes max(m1,1), and $m2^-$ and $m2^+$ are defined similarly. E1—(m1,M1)—⟨R12⟩—(m2,M2)—E2 is rearranged to:

E1—($m1^-$,1)—⟨R13⟩—($m2^+$,M2)—E3—($m1^+$,M1)—R23—($m2^-$,1)—E2

(c) If R12 had an inclusion constraint E2[A2] ⊃ E1[A1], impose the inclusion E3[A3] ⊃ E1[A1]. Otherwise, impose the (weaker) constraint that all nonnull values in (E1[A1] ∪ E2[A2]) appear in E3.

(d) Create R23 between E2 and E3 analogously to the creation of R13.

(e) Infer additional constraints based on key or "mandatory" constraints on E1 and E2.

PROOF. We show that steps (a)–(d) produce a rearrangement. Adding an additional rearrangement (step (e)) to a rearrangement still yields a rearrangement.

The instance mapping I(s1) (as illustrated in Section 3.6) creates an e3 instance with value e3[A3] for each nonnull value that appears in both

E1[A1] and E2[A2]. The populations of R12 and R13 are established by value determination. Other entities and relationships are unchanged. The inverse I^{-1} deletes E3, R13, and R23 and reinserts the value-determined relationship R12.

Total. The null-not-allowed constraints on E3 hold, since e1 instances were copied directly. Inclusions hold, since only values appearing in both E1[A1] and E2[A2] were copied. The populations of R13 and R23 were defined by the value-determined constraints and so must satisfy them. Furthermore, if E2[A2] $\supset$ E1[A1], then all values from E1[A1] generated tuples of E3, so E3[A3] $\supset$ E1[A1] applies.

Now we must verify the participation constraints. We verify constraints on R13; the proof for R23 is analogous.

—Verification of min_participation $\geq m1^-$: If $m1 = 0$, $m1^- = 0$, and there is no constraint. As long as $m1 > 0$, e1 is R12-related to at least one e2; hence there is an e3 such that e1[A1] = e3[A3].

—Verification of "max_participation ≤ 1": A3 is a key of E3, so only one e3 can match the value e1[A1].

—Verification of "min_participation $\geq m2^+$": For each instance e3, there is at least one instance e2 such that e3[A3] = e2[A2]. By the minimum participation on E2, at least m2 instances of E1 match this value. At least one matches, since e3 instances are created only when there are matching values in both.

—Verification of "max_participation $\leq$ M2": For each e2, at most M2 instances of E1 can have e1[A1] = e2[A2]. Hence at most M2 instances can match the corresponding e3.

Surjective. Suppose s2 is an instantiation of S2. We show that $I^{-1}(s2)$ satisfies the constraints on s1 and that I^{-1} is an inverse.

—Proof of constraint satisfaction: Since I^{-1} populates R12 by value determination, R12 is a relationship satisfying the value-determined constraint.

—Proof that *inclusions* are satisfied: We consider the case where an inclusion on R12 generated the constraint E3[A3] $\supset$ E1[A1] on S2. Since E2[A2] $\supset$ E3[A3] we have E2[A2] $\supset$ E1[A1] in s2. Since I^{-1} does not alter the populations of E1 and E2, the constraint will also apply to $I^{-1}(s2)$.

—Proof that *participations* are satisfied: In S2, the number of e3 for each E1 is at least $(m1^- * m1^+)$ and at most (1*M1). A case analysis (m1 zero and nonzero) shows that the first expression equals m1. Hence the constraints (m1,M1) are satisfied.

—Proof that $I(I^{-1}(s2)) = s2$, i.e., that I^{-1} is an inverse: When I creates R3, the population has one instance for each value matched between E1[A1] and E2[A2]. The extra constraint introduced at step (c) says that any population of S2 must have exactly that population for R3. (Inclusion would imply this constraint.)

Injective. Suppose distinct instantiations of S1 (denoted s1 and s1′) are mapped to the same instantiation s2 of S2. For every object x other than R12,

I does not change the population. Hence s1[x] = s2[x] = s1′[x]. For R12, the population is value determined; since E1 and E2 have identical populations in s1 and s2, R12 must be identical also. Hence s1 and s2 have identical populations everywhere □

ACKNOWLEDGMENTS

Victor Markowitz's challenging comments encouraged us to clarify the justification of ER+ .

Our heartfelt thanks to out to our colleagues on the DDEW project—Gretchen Brown, Mark Friedell, David Kramlich, John Lehman, Richard McKee, Penny Rheingans, and Ronni Rosenberg. This team designed, built, tested, documented, and handed off DDEW to the client within an incredibly productive and intellectually stimulating two-year time span.

REFERENCES

BATINI, C., LENZERINI, M., AND NAVATHE, S. B. 1986. Comparison of methodologies for database schema integration. *ACM Comput. Surv. 18,* 4 (Dec.), 323–364.

BISKUP, J., AND CONVENT, B. 1986. A formal view integration method. In *Proceedings of the ACM SIGMOD Conference.* ACM, New York, 398–407.

CASANOVA, M., AND AMAREL DE SA, J. 1984. Mapping uninterpreted schemes into entity-relationship diagrams: Two applications to conceptual schema design. *IBM J. Res. Dev. 28,* 1 (Jan.).

CASANOVA, M., VIDAL, V. M. P. 1983. Toward a sound view integration methodology In the *ACM Conference on Principles of Database Systems.* ACM, New York, 36–47.

COSMADAKIS, S., AND KANELLAKIS, P. 1984. Functional and inclusion dependencies: A graph-theoretic approach. In the *ACM Conference on Principles of Database Systems.* ACM, New York, 29–37.

D'ATRI, A., AND SACCA, D. 1984. Equivalence and mapping of database schemes. In *Proceedings of the 10th Conference on Very Large Databases* (Singapore). VLDB Endowment, 187–195

DIEDERICH, J., AND MILTON, J. 1988. New methods and fast algorithms for database normalization. *ACM Trans. Database Syst. 13,* 3 (Sept.).

HULL, R. 1984. Relative information capacity of simple relational database schemata. In *Proceedings of the ACM Conference on Principles of Database Systems.* ACM, New York, 97–109.

JOHANNESSON, P., AND KALMAN, K. 1989. A method for translating relational schemas into conceptual schemas. In *Proceedings of the 10th International Conference on the Entity-Relationship Approach* (Toronto, Oct.), E/R Institute.

KALINICHENKO, L. A. 1990. Methods and tools for equivalent data model mapping construction. In *Proceedings of the International Conference on Extending Database Technology—EDBT'90* (Venice). Springer-Verlag, New York, 92–119.

KLUG, A. 1979 Entity-relationship views over uninterpreted enterprise schemas. In *Proceedings of the International Conference on the Entity-Relationship Approach* (Los Angeles, Dec.), E/R Institute, 39–59.

LING, T.-W. 1986. An analysis of multivalued and join dependencies based on the Entity-Relationship approach. *Data Knowl. Eng. 1,* 253–271.

LING, T.-W. 1985. A normal form for Entity-Relationship diagrams. In *Proceedings of the 3rd International Conference on the Entity-Relationship Approach,* E/R Institute.

MAIER, D 1983. *The Theory of Relational Databases.* Computer Science Press, Rockville, Md.

MANTHA, R. 1987. Logical Data Model (LDM): Bridging the gap between ERM and RDM. In *Proceedings of the 6th International Conference on the Entity-Relationship Approach,* E/R Institute, 42.

MARKOWITZ, V. 1990. Referential integrity revisited—An object-oriented perspective. In *Proceedings of the 16th Conference on Very Large Databases* (Brisbane, Australia). VLDB Endowment.

MARKOWITZ, V., AND SHOSHANI, A. 1989. On the correctness of representing extended Entity-Relationship structures in the relational model. In *Proceedings of the ACM SIGMOD Conference* (Portland, Ore., June). ACM, New York.

NAVATHE, S., ELMASRI, R., AND LARSON, J. 1986. Integrating user views in database design. *IEEE Comput. 19*, 1 (Jan.).

OERTLY, F., AND SCHILLER, G. 1989. Evolutionary database design. In *Proceedings of the 5th International Conference on Data Engineering* (Los Angeles). IEEE, New York.

RAM, S. 1993. Automated tools for database design: State of the art. CMI Working Paper, Dept. of MIS, College of BPA, Univ. of Arizona, Tucson, Ariz.

REINER, D. 1992. Automated database design tools. *Can. Inf. Process.* (July/Aug).

REINER, D. 1991. Database design tools. In *Conceptual Database Design: An Entity Relationship Approach,* C. Batini, S. Ceri, and S. Navathe, Eds. Benjamin Cummings, Menlo Park, Calif.

REINER, D., AND GONZALES, D. 1985. *User's Guide for the Database Design and Evaluation Workbench (DDEW).* DDEW-SD-835-2, Computer Corporation of America, Cambridge, Mass.

REINER, D., BROWN, G., FRIEDELL, M., LEHMAN, J., MCKEE, R., RHEINGANS, P., AND ROSENTHAL, A. 1986. A database designer's workbench. In *Proceedings of the 5th International Conference on the Entity-Relationship Approach* (Dijon, France).

ROSENTHAL, A., AND REINER, D. 1989. Database design tools: Combining theory, guesswork, and user interaction. In *Proceedings of the 8th International Conference on Entity-Relationship Approach* (Toronto, Oct.), E/R Institute.

ROSENTHAL, A., AND REINER, D. 1987. Theoretically sound transformations for practical database design In *Proceedings of the 6th International Conference on the Entity-Relationship Approach.*

SEGEV, A. 1987. Transitive dependencies. Surveyor's Forum. *ACM Comput. Surv. 19*, 2 (June).

SOCKUT, G., AND MALHOTRA, A. 1988. A full-screen facility for defining relational and Entity-Relationship database schemas. *IEEE Softw.* (Nov.).

TEOREY, T. 1990. *Database Modeling and Design: The Entity-Relationship Approach.* Morgan Kaufmann, San Mateo, Calif.

TEOREY, T., YANG, D., AND FRY, J. 1986. A logical design methodology for relational databases using the extended Entity-Relationship model. *ACM Comput. Surv. 18,* 2 (June).

Received July 1991; revised March 1993; accepted July 1993