Tools and Transformations – Rigorous and Otherwise –  
For Practical Database Design

Arnon Rosenthal¹  
Xerox Advanced Information Technology  
Four Cambridge Center  
Cambridge, MA 02141  
617-499-4445  
arnie@xait.xerox.com

David Reiner  
Kendall Square Research Corporation  
170 Tracer Lane  
Waltham, MA 02154  
617-895-9449  
reiner@ksr.com

Abstract

We describe the tools and theory of a comprehensive system for database design, and show how they work together to support the design process. The Database Design and Evaluation Workbench (DDEW) system uses a rigorous, information-content-preserving approach to schema transformation, but combines it with heuristics, guesswork, and user interactions.

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, 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 our approaches for identifying redundant relationships and for an interactive view integrator. Lastly, we examine the relevance of database theory to building these practically-motivated tools, and contrast the paradigms of system builders with those of theoreticians.


General Terms: Design, Heuristics, Theory

Additional Keywords and Phrases: Database design, normalization, view integration, design heuristics, applications of database theory, data model translation, entity-relationship model, computer-aided software engineering

Submitted: July 30, 1991

¹ Part of this work was done while this author was at ETH-Zurich.
1 Introduction

The Database Design and Evaluation Workbench is a graphics-oriented database design system prototype built on Jupiter and Sun workstations at CCA Advanced Information Technology. In this paper, 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. In both representations and design operations, we exploited 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 graphic 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. In addition, several mechanisms are provided to help the designer to cope with complexity and with large designs. These include highlighting of arbitrary affinity groups of related objects, the use of color to identify missing or dubious information, 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.

For a broad overview of DDEW, including the user interface, graphical display of progress, design methodologies, and implementation considerations, see [REIN86]. In this paper, we concentrate on three of the unusual aspects of the system's data models and tools: (1) a unified underlying data model, (2) content-preserving schema transformations, and (3) heuristic tools for initial design.

1.1 A unified underlying data model

One single model (an extended ER model called ER+) provides the internal representation, transformations, and editing operations for designs at both conceptual and logical stages, and over all of our target logical models. Building the system over this unified 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 all of the 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.

1.2 Content-preserving schema transformations

As design progresses, details are accumulated, errors are fixed, and the schema is 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 reasons of implementability (to use only the structures permitted in a target DBMS), performance, or convenience (so that schema objects will match organizational units or existing definitions).

A design system ought to ensure that a transformation between equivalent schemas will not 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 depends on the designer’s final check, instead of on the union of all accuracy checks in 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.

---

1 DDEW was supported by Rome Air Development Center, USAF, under Contract F30602-83-C0073 to Computer Corporation of America (CCA). CCA's Advanced Information Technology Division is now Xerox Advanced Information Technology.

2 A slight variant of this notation was adopted in [TEOR90].
As formalized below, the information content (or capacity) 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 [HULL84]) 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 modules which preserved the information content of the schema.

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 suggest or carry out schema transformations that are justified as being possible improvements, rather than as being rigorously correct. New information can be added profitably by tools with built-in assumptions and defaults. 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 is intended as a reference for the reader to understand how DDEW’s tools and transformations fit in the context of the general design methodology. 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.
<table>
<thead>
<tr>
<th>Step</th>
<th>Objects Accessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Analysis</td>
<td>English description</td>
</tr>
<tr>
<td></td>
<td>Functional dependencies (FDs)</td>
</tr>
<tr>
<td></td>
<td>Transactions (against attributes)</td>
</tr>
<tr>
<td>Conceptual Design</td>
<td>FDs and transactions</td>
</tr>
<tr>
<td></td>
<td>ER schema diagram and transactions against it</td>
</tr>
<tr>
<td>Logical Design</td>
<td>ER schema diagram</td>
</tr>
<tr>
<td></td>
<td>Relational, network, or hierarchical schema diagram and transactions against it</td>
</tr>
</tbody>
</table>
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, focussing on questions of comprehensibility, robustness, and graceful degradation. Section 6 gives conclusions. Preliminary versions of this work appeared in [ROSE87a] and [ROSE89].
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 particular data model or DBMS interface), and physical schemas that capture implementation detail. Each of these data models must be representable within the database design system, either as a separate model or as a special case of a more general model.

We took the latter approach. One model, called ER+, provided the internal representation and semantics for all conceptual and logical design activities. As a result, the same tool code could be run under many different circumstances. This shortened the learning curve for database designers running DDEW, not to mention minimizing our implementation effort in building the system.

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 [CHEN76]. Its constraint constructs are numerous and rather general; they encompass most of the constraints of the classical data models, plus handling nulls. From a theoretical standpoint, the contribution of ER+ is limited to two areas – the treatment of nulls in inclusion constraints (done to facilitate optional relationships), and the explicit treatment of arbitrary sets of attributes that can determine a relationship.

The constraint constructs were selected to support our set of design transformations. In models without constraints, content-preserving transformations can do no more than (trivially) reorder attributes [HULL84]. In other words, constraints are required if meaningful transformations are to be content-preserving.

ER+ begins with conventional entities, attributes, and binary relationships 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 are discussed briefly in [REIN86]. The ER+ constraint constructs most relevant to initial design are listed below.

Notation

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. 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.

Constraints

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.

Null-not-allowed: This constraint forbids an attribute value to be null in any instance of the entity type. We allow the constraint to be generalized to multiple attributes, requiring that at least one attribute in the set be non-null. The generalization allows (First_Name, Middle_Name, Last_Name) to be null-not-allowed and a primary key, even though Middle_Name can be null.

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 to show (min,max) participation by the entities on each side of relationship R. Ignoring minimum, we loosely say that R is M2:M1. “Unknown” is a legal value for min and max participation values. DDEW highlights places where more information is needed, yet allows design to proceed based on partial information.

Relationship parent: For a 1:n relationship, the parent is the entity on the 1 side. For a 1:1 relationship, a designer may declare which entity should be treated as the parent when creating a hierarchical schema from a network schema. This very specific constraint was created to help resolve an otherwise ambiguous schema transformation issue when producing a hierarchical schema.

Value-determined relationship: R is value-determined by matching attributes A and B if and only if (R-related entity instances have identical nonnull values for the matching attributes, and entity instances with identical nonnull values for E1[A] and E2[B] are R-related). That is, the relationship R consists exactly of the set of entity pairs (e1,e2) such that e1[A]=e2[B].

The matching attributes in a so-called foreign key and a primary key value-determine a relationship between the corresponding entities. For example, the value-determined relationship HAS_EMP between DEPT and EMP is
determined by the rule "a DEPT instance d will be related to all EMP e such that d[Dept#] = e[Dept#]". Speaking loosely, we say that the "same" attribute Dept# appears in both entities; actually, it is values of the attribute that are replicated.

ER+ does not require that relationships be value-determined, but value-determined relationships are very useful. First, they associate attribute values with relationships, e.g., in a relational-style schema. Second, they express a constraint on attributes (not necessarily keys\(^\text{1}\) ) that are replicated in related entities. Finally, analysis of the determining attributes can reveal whether one relationship is the composition of others (see section 4.2.2).

**Inclusion:** E1[A] includes E2[B] (denoted E1[A]...E2[B]) if everynonnull value e2[B] appears as the value of some e1[A]. When a value-determined relationship is mandatory, an inclusion constraint can be inferred automatically.

**Multiset-relationship versus set-relationship:** A multiset relationship's population is a multiset of ordered pairs; a set-relationship (or just relationship) allows no duplicates. Multiset relationships are useful in defining rearrangement modules, but the distinction seemed to us too fine to present to designers (especially since ER+ relationships don't have attributes).

**Set-key:** 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. This construct was included primarily to model uniqueness constraints within a Codasyl set.

Each constraint is expressible as a predicate over a well-determined set of objects (entities, relationships, and attributes). Rearrangement transformations can rarely delete an object that has a non-redundant constraint, since deletion of the constraint would usually change the schema's information content.

**Constraints as Logic Formulas**

It is natural to ask what role a general language like logic should play in expressing constraints. Logical expressions can be executed as predicates, or manipulated by a theorem prover, e.g., when devising or verifying rearrangement algorithms. But logic is not appropriate as the immediate basis for transformations or user interfaces, which must operate on higher level concepts. For example, a constraint declaration Key(E,K) is more appropriate for both purposes than the full logic assertion

\[
\text{Instance}(e, E) \text{ and } \text{Instance}(e', E) \text{ and } \text{equal}(e[K],e'[K]) \implies \text{equal}(e,e')
\]

Substantial computation time and software development would be needed to process low-level logical expressions and translate the results back to the user; working at this level would slow the system considerably.

### 2.2 Using Style Subsets of ER+ for Schema Conversion

Ideally, a database design system should be able to capture, display, and transform schemas in multiple models. Multimodel support leverages the investment in workbench software, helps users to combine information from multiple sources, and – if done in an open fashion – allows 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.

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, including the interchange formats used in existing CASE systems, IDEF1-X format for engineering, and the IRDS and AD/Cycle Repository formats for data dictionaries. In the future, support for inheritance, k-ary relationships, and object-oriented constructs may become increasingly important.

A style subset of ER+ is the set of constructs of the unified model corresponding to a given target logical model. In the network style, relationships are 1:1 or 1:n, they must connect distinct entity types, and multisets (duplicates) are prohibited.\(^2\) The hierarchical style is considered a strict 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 and relationship multisets are prohibited. We remark in passing that the “stricter” relational model style of [MARK89] can be captured by imposing further constraints that all attribute values are nonull, and that every value-determined relationship includes a key of at least one entity.

---

\(^1\) Occasionally, m:n value-determined relationships on non-key attributes are useful (e.g., between Warehouses and Customers with the same City_Name).

\(^2\) Repeating groups were not modeled in ER+.
Arbitrary ER+ Schema

Rearrange to a Schema in the target style
Defining style subsets of ER+ for relational and network models allowed 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). However, the two 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 second step is to perform a purely syntactic translation to the particular DDL; constructs that cannot be translated are converted to comments on the resulting schema. In translating from relational-style ER+ to relational DDL, constructs converted to comments include relationships (now carried by attribute pairs), relationship names, minimum and maximum participation, and referential integrity.

The net effect is that the style-subset representation of ER+, not the more limited DDL, is the full description of the logical database design. Any constraints in this representation that the DBMS does not handle need to be enforced by application programmers. 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.

Producing hierarchical-style logical schemas was done slightly differently, with the starting point being a network-style schema. The hierarchical-style subset is not fully expressive, since equivalence can be lost when multi-parent entities are reduced to single-parent ones or cycles are arbitrarily broken (see section 3.7).

### 2.3 Benefits of a Unified Underlying Model

ER+, DDEW’s unified underlying model, can be regarded as a collection of abstract data types, to be used in implementing the various user-visible models. 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. Building over ER+, brought considerable benefits as outlined below.

#### 2.3.1 Reusable Constructs and Operations

Major rearrangements within and between style subsets were written as compositions of small ER+ rearrangement modules. Both the theory (i.e., detailed proofs of equivalence) and the code for these modules were reusable in several contexts. Nearly every ER+ feature is used in several visible models, greatly reducing development effort.

**Reuse of constructs:** Entities in ER+ are used to implement relations, entities, and record types, all of which involve an aggregate of attributes. Relationships in ER+ provide an underlying abstraction that is useful for semantic connections or carriers of constraints among entities, relations, or record types. Attributes are ubiquitous among the visible models. Constraints and datatype declarations are of interest to the conceptual designer and also in designing logical and physical schemas. The same constructs can be used for either semantic modeling or in a logical data model that is a direct abstraction of a DBMS interface (see section 2.3.6).

**Reuse of operations:** For example, copying attributes from one entity into a related one is important for generating keys in inter-model translation, and for producing useful redundancy in a physical or logical schema. 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.

Figure 3a shows the spaghetti-like potential for redundancy and inconsistency with independent models and translators. In contrast, Figure 3b shows the simplicity of a single underlying model.

---

1 DDEW was built before widespread support for referential integrity within relational DDL.
Conceptual model C

Logical model L1

L2

L3

...
Opportunities for reuse are more limited – but still real – even in a more narrowly-targeted workbench. [MARK89] 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. This was accompanied by an extensive theory describing 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 inter-model correspondences over a single formal system would factor out operations (such as attribute copying and deletion) that appear in both inter-model and intra-model transformations.

2.3.2 Major Schema Changes Expressed within the Formal System
All significant schema changes are expressed within the formal system. ER+ has reusable modules and is much more amenable to proofs than, say, large transformations written in C.

A good rule of software design is that a tool should change content (e.g., move attributes and add constraints) or change representation (e.g., transform diagrams to text) but not both. Specifically, we do not generate SQL DDL directly from an arbitrary ER schema, because the translation would require substantial content changes, such as migration of key attributes and introduction of new constraints. Instead, we first produce an ER diagram whose constructs match the relational model fairly closely, and then do 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). When re-engineering a relational schema we create an entity for each relation, having the attributes of the relation; a referential integrity constraint becomes a relationship with an inclusion dependency. Even if a source schema has redundancies, is missing constraints, or is otherwise “bad,” it is still brought directly into ER+ before additional transformations and editing take place. An alternative would be to translate all schemas into the relational model, transform them using relational theory, and then translate them back. However, translation into the relational model would lose ancillary information about a design (e.g., its diagrammatic layout, textual annotations, and arbitrary constraints). It is quite difficult to recover this information or to specify how it should be carried through relational transformations.

2.3.3 Flexibility in Information Capture
ER+ can capture information at all levels of details simultaneously during design, although the screen displays and instructions to the designer do impose restrictions. In contrast, a system that uses separate models cannot, during conceptual design, capture information that the models consider strictly logical (e.g., details of constraints and attributes). The single-model approach thus enhances workbench customizability, and minimizes repetitive interviews with end-users and multiple passes through existing documentation.

Since ER+ can express schemas that conform partly to one style, partly to another, gradual schema evolution is possible. For example, if a workbench has no translator to a target physical model, it is possible to use a higher-level schema as an initial design. The "mixed schema" capability also facilitates building tools that degrade gracefully when confronted by unfamiliar constructs (e.g., due to model extensions). As long as all schemas are expressed in ER+, an offensive construct can be passed through into the output (and flagged for user attention).

2.3.4 Complexity of 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 (e.g., CODASYL repeating groups or IMS hierarchies), and physical storage and indexing constraints.

We depended on three tactics to deal with this complexity, involving both the user interface and the design tools of DDEW.

(1) Constraints are generally viewed graphically within pop-up windows that limit and focus the information seen by the designer. In addition, DDEW customizes its logical-design displays somewhat 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 that fact. [MANT87] carries these ideas further, allowing the user to dynamically adjust the information shown in pop-up windows on the display.

(2) We made it possible to extend the constraints in the model without completely invalidating the tools. Tools that detect an unfamiliar constraint (either a new construct, or a logic formula) that references an object are prevented from changing the referenced object (entity, relationship, or attribute). For example, attribute deletion (see section 3.3) will delete attributes that value-determine a relationship only if the attributes are not mentioned in other constraints. This approach permits tools to run unmodified after the underlying data model has been changed, and also lends itself to incremental and concomitant tool evolution.
(3) We provided tools to generate or remove redundant declarations. This was useful in the few cases where we were unable to factor out the semantic overlap between user-visible models. For example, when a value-determined relationship is declared mandatory, an inclusion declaration is unnecessary, but for optional relationships omitting the inclusion construct would have lost expressive power. Dealing explicitly with redundant declarations was especially helpful in uncluttering diagrams which had redundant relationships (see section 4.2.2).
3 Rearrangement Transformations in DDEW

A rearrangement is a transformation that preserves the content of a schema. With rigor as a goal, we modified several existing schema transformation algorithms to make them into rearrangements. Some were small, modular building blocks (e.g., the transformation that copies a single attribute from one entity into another); some were large tools (e.g., the ER-to-relational translator). The large tools were built as sequences of modular, reusable rearrangements that were independently verifiable. The composition of rearrangements is a rearrangement (see Appendix).

Each of the transformations below is defined for a single relationship and its incident entities. In general, a DDEW command applies several different transformations. Each transformation is considered for all relationships in the design.

Section 3.1 motivates and 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, transforming a relationship into an entity, creating primary keys, and data model style translations. All the transformations described have been implemented; however, the descriptions incorporate improvements added afterward. We briefly describe the motivation for each rearrangement and then present the formal transformation. Each transformation includes preconditions – no change occurs if these preconditions fail. The appendix contains verifications that some of the transformations are content-preserving; the remaining proofs appear in [ROSE87b].

Deliberately, we imposed few of the restrictions that simplify algorithms in the theoretical literature. Assumptions like “no null values” or “each entity has only one key” or “every relationship connects distinct entities” are violated by too many real schemas.

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)|eEMP', dDEPT, e[Dept#]=d[Dept#]} (2) EMP'[Dept#] is non-null
     DEPT[Dept#] ... EMP'[Dept#]

In many 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.

Definitions

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 any two schemas, and let s1 and s2 refer to instantiations of S1 and S2 respectively. A mapping 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, satisfies four conditions:
1. total (for each schema instantiation 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 [HULL84]) is that the transformation is invariant under permutation of nonnull attribute values. Without this
condition, every schema $S_1$ would be equivalent to a degenerate schema consisting of a single integer attribute taking values between 1 and the number of instantiations of $S_1$.

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.”

### 3.2 Attribute Copying Transformations

These transformations copy a set of attributes from one entity $E_1$ into an $R$-related entity $E_2$, and adjust constraints appropriately. One reason to copy attributes is to represent the relationship $R$ by importing a foreign key (of $E_1$) into $E_2$ (see section 3.6). Another reason is to speed queries that need both the copied attributes and the original attributes of $E_2$.

We restrict our attention to cases where the constraints on the resulting schema are expressable with ER+ constructs. The preconditions of the transformations exclude situations (such as copying part of a key, or copying across m:n relationships) where it would be necessary to specify additional logical predicates to obtain equivalence. For simplicity, we assume that attribute names are unique; the actual DDEW implementation detects conflicts and generates new names.

#### Key copying

An entire key may be copied across a non-value-determined relationship to make the relationship value-determined. This transformation was illustrated informally in section 3.1.

- **Copying a key $A_1$ of $E_1$ across a non-value-determined relationship $R$ from $E_1$ to $E_2$:***
  - Preconditions for applicability: $R$ has at most one $E_1$ for each $E_2$, and is non-value-determined. $A_1$ contains a key of $E_1$ (i.e., a key or a superset of a key). $E_1[A_1]$ is null-not-allowed.
  - Result: Attributes $A_1$ are added to $E_2$. $R$ has the constraint that it is value-determined by $A_1$. $E_1[A_1]$…$E_2[A_1]$. If $R$ was mandatory from $E_2$, then $E_2[A_1]$ is null-not-allowed.

#### Additional attribute copying

- **Copying attributes $A_3$ across a value-determined relationship $R$ from $E_1$ to $E_2$:***
  - Preconditions for applicability: $R$ has at most one $E_1$ for each $E_2$, and is value-determined with $E_1[A_1]$ matching $E_2[A_2]$. $A_1$ contains a key of $E_1$.
  - 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.

Key removal is useful in going from relational-style schemas to schemas with information-bearing relationships. Relational schemas may arise from existing databases and files, or from normalization (section 4.1). After relationships are inferred and checked (see section 4.2,) one has a schema with value-determined relationships. To make this a purer ER schema, referencing attributes are removed.

There are at least two ways that a schema may include redundant attributes not needed for referencing. First, the schema might have been created by integrating separate user views. Second, the redundancy may be a performance optimization. In either case, the redundancy should not be reflected in a conceptual schema. 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.

### 3.4 Inferring Constraints for Value-Determined Relationships

The rearrangements in this section infer constraints that are implied by other constraints. These inferences are made in several tools. The rearrangements can be reversed, removing constraints to produce a minimal set. Minimal sets impose less run-time overhead, and are sometimes easier for other transformations to handle.

---

1. Such physical-design decisions typically must be made during design of the logical schema. Unfortunately, many constructs that are considered “logical” – because they are visible at the DBMS interface – also carry physical implications in most DBMSs.
In the rules below, R12 denotes a *value-determined* relationship between E1 and E2, based 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 set minimum participation of E2 in R12 to 0.

Although this declaration imposes no constraint, we still find it worthwhile. DDEW prefers explicit denials to omission, since the denial is evidence that the question has been examined. In fact, when participation information has not been supplied, the the corresponding half of the relationship diamond on the screen is displayed in an alarming red color.

### 3.5 Transforming a Relationship to an Entity

The decision of how to model a connection often needs to be changed, typically in the direction from relationship to entity. For example, an ER+ relationship must be transformed to an entity if the designer wishes to attach attributes. Conversions also occur when m:n relationships need to be represented in a style subset that lacks that construct.

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

#### Multiset relationship to entity

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

Example: Suppose USED_SKILL relates EMP and SKILL entities. (A pair in USED_SKILL may be created each time an employee uses a given skill on one of the projects he works on; hence the multiset.) The rearrangement might be:

```
EMP---(1,n)---<USED_SKILL>---(0,n)---SKILL  is rearranged to:  EMP---(1,n)---<R13>---(1,1)---USED_SKILL---(1,1)---<R23>---(0,n)---SKILL
```

The relationship <USED_SKILL> is replaced by the entity USED_SKILL and the two new relationships <R13> and <R23>. Note that the new entity USED_SKILL has no attributes. Later steps (part of a larger rearrangement) can copy in some attributes, or create a surrogate attribute as key.

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)---<R13>---(m2,M2)---E2 is rearranged to:  E1---(m1,M1)---<R13>---(1,1)---E3---(1,1)---R23---(m2,M2)---E2
```

#### Modification to deal with a set relationship

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

To express the set constraint, attributes must be copied into the derived entity. It is not clear whether the improvement in schema quality justifies the complexity. For this example, suppose <USED_SKILL> is a set relationship, so that a given (EMP, SKILL) pair can appear at most once. The key of the new USED_SKILL entity will be the union of keys from EMP and SKILL.

We compose several other rearrangements to handle the set-relationship constraint. The steps are:

a. Identify primary keys K1 for E1, K2 for E2 (see section 3.6).
b. Execute the multiset rearrangement (above).

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

d. Declare (K1•K2) a primary key of E3.

Converting a value-determined set relationship

• Converting a value-determined set 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 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 a salesman. 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.

\[ E1\text{---} (m1, M1) \text{---}<R12>\text{---} (m2, M2) \text{---}E2 \text{ is rearranged to: } \ E1\text{---} (m1−, 1) \text{---} <R13>\text{---} (m1+, M1) \text{---}R23\text{---} (m2+, M2)\text{---}E2 \]

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 (see section 3.4).

Converting a value-determined, multiset relationship

• Converting a value-determined, multiset relationship R12 (between entities E1 and E2) into a new entity and two new relationships.

We expect this case to be very rare, and therefore have not done a detailed study. It appears sufficient to modify the basic algorithm to copy all matching attributes instead of just key attributes. The relationship is not really determined by the attribute values, since those values cannot determine the number of copies.

3.6 Rearrangements Involving 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).

\[[\text{If } E1 \text{ has maximum_participation = 1 in } R12, \text{ then}]\]

1. Copy K1 into E2 and declare copied attributes K2 a key of E2;

2. 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;

3. Else (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 consider this to be making visible an attribute that was already implicitly present, so that it does not affect the information content.

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. We suspect that the only practical way to produce complex rearrangements (like these) is by composition of smaller rearrangements. Otherwise it may be too difficult to get the constraints correct, and to verify invertibility of the instance mapping. For completeness, we also describe the transformation from a network-style to a hierarchical-style subset; this is not a content-preserving rearrangement since information may be lost in going to the weaker model.

### 3.7.1 Transformation to Relational Style

a. Obtain keys for each entity (see section 3.6).
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 multiset relationships, and to replace m:n relationships that are not value-determined.

### 3.7.2 Transformation to Network Style

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

### 3.7.3 Transformation from Network to Hierarchical Style

Precondition for applicability: Begin with a network-style schema.
a. Declare an arbitrary parent constraint if missing for a 1:1 relationship.
b. Reduce each multi-parent entity to a single-parent one by arbitrary deletion of all but one of the relationships in which it is the child.
c. Remove any cycles of length greater than 1 by arbitrary relationship deletion within the cycle.
d. Determine “root” entities having no parents.
e. Create a SYSTEM parent node, and attach all root entities to it via 1:n relationships to form a single hierarchy.
f. For all relationships, if any child-parent link shows partial participation, convert it to mandatory participation.

For obvious reasons, this 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 multi-parent entities, and to break cycles if present – before invoking the transformation. In retrospect, a better approach might have been to insert an initial step (before step) a as follows:

a’. Choose a maximal set H of relationships whose graph contains no cycles. Convert relationships not in H to be value-based. Apply steps a-f only to relationships in H.

This would obviate the need for deletion of arbitrary relationships in steps b and c.
4 Heuristic Tools that Augment Schemas

In DDEW, manual editing or schema input utilities are used to produce an initial design. This design may be either an entities-only schema, without relationships; an ordinary schema, with entities, relationships, and screen layout information; or several existing schemas that eventually need to be integrated. A collection of functional dependencies falls within the first category, since a dependency can be treated as an entity with a key declaration. 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 rigor (i.e., than preserving content-equivalence). In this section we show how heuristic tools that make likely guesses can be combined with transformations and user input to rapidly improve an initial specification.

Aided by heuristic tools, the designer has several options for improving an initial design. He or she may:

- 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, if present (Section 4.3).

4.1 Normalization for Redesign of Existing Schemas

This section examines how normalization techniques, originally 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. The normalization process takes information through the following stages:

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.

Stages 2-4 appear when relational schemas are synthesized from functional dependencies. The traditional rigorous approach to normalization cannot provide a design tool that manipulates arbitrary ER+ schemas, for three reasons. First, normalization theory is not adequate to handle attributes that may be null, or both functional and inclusion dependencies on arbitrary relations. These problems are solvable, but in ways that impose severe limitations on the user. For example, one can transform the schema to one without nulls – at the cost of a major database reorganization and an increase in the number of entity types. It is possible to handle inclusion and functional dependencies on schemas that are “ER-compatible” [MARK89], but then normalization cannot be used directly on a schema obtained from existing relations or from file definitions. Second, normalization does not handle the problem of mapping relational schemas back to ER+. Third, the universal relation scheme assumption [MAIE83] (denoted URSA) cannot be relied on. Certainly, few corporate data administrators can tell us whether it holds.

Below, we discuss the pragmatic effects of various approaches to normalization. Section 4.1.1 compares approaches that normalize each entity separately with approaches that deal with functional 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, including entities, relationships, and diagram layout.

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 applicable to each entity, and performs a separate normalization. If there are relationships in the schema, dependencies may be obtained from key constraints and from 1:1 value-determined relationships [LING85]. Each entity does indeed satisfy URSA, so normalization preserves all dependencies.

We suspected that local normalization will make little or no improvement in most designs, so we did not implement it. Since the algorithm works one entity at a time, it cannot detect redundancy among entities. Also, we did not expect many non-key functional dependencies. Naive users do not understand them, and when a sophisticated user discovers a non-key dependency, he or she may split the entity through manual editing.

Global Normalization throws functional dependencies from all entities into a “soup of attributes,” and uses normalization to simplify the ensemble. For the duration of normalization, attributes of the same name in different entities are considered identical (i.e., we temporarily adopt URSA). Since the assumption is dubious, the results of
normalization are untrustworthy – in DDEW, they are treated as suggestions to the designer. For example, we leave it up to the designer to decide whether to merge entities with identical keys, or where incident relationships should be reconnected after an entity has been split.

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 and non-synonyms specified by the user, or from constraint information (inclusion dependencies, and matches between attributes in value-determined relationships).

4.1.2 Preserving the Input Schema

An existing input schema has relationships, constraints, names, and diagram layout; normalization yields just entities and their functional dependencies. It is essential to construct schemas that combine the original information with the suggestions from the normalizer. Approaches to this problem differ in the development effort to be devoted specifically to reconstruction after normalization. If the goal is to minimize special-purpose development, one obtains a set of functional dependencies and uses a traditional algorithm without modification,to produce an entities-only result schema. The view integrator combines this result with the original input, under a regime that gives priority to names and display information from the original schema. Since view integration is also used for other purposes, one can justify more effort in implementing its heuristics and user interactions.

The above approach requires view integration to make its decisions with no help from the normalization algorithm. A simple extension to normalization is to tag each dependency with its source entity-name. An entity in the resulting entities-only 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 might be allocated based on the fate of attributes involved in value-determination constraints. Since diagram readability is important, one might attempt to position the entities produced by the split together in the diagram.

4.2 Relationship Heuristics

Schemas need relationships – a display of forty 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 existing technology in several ways. 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. Finally, we introduce new, very general rules for identifying redundant relationships. However, unlike [CASAS84], our algorithm is not content-preserving, since it adds information by guessing likely relationships and inclusions. Also, unlike [JOHA89], 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:

<table>
<thead>
<tr>
<th>Entity</th>
<th>primary_key DEPT(Dept#, Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dept# EMP(St#, Dept#, Name, Age)</td>
<td></td>
</tr>
<tr>
<td>SS#PROJECT(Proj#, Dept#, Budget)</td>
<td></td>
</tr>
<tr>
<td>Proj#TASK(Task#, Name, Proj#, Due_Date)</td>
<td></td>
</tr>
<tr>
<td>Task#</td>
<td></td>
</tr>
</tbody>
</table>

(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.

\[1\] In DDEW, schema reconstruction was not implemented. Instead, the normalization-of-existing-schemas tool simply reported changed dependencies for the designer to act on.
A decision to create a relationship determined by either non-key attributes or by multiple matching attributes ought to be reviewed by the designer. With non-key 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. Detection of matching attributes could have been made more intelligent. For example, one could identify matches based on knowledge of prefixes and suffixes (e.g., Dept, Dept#, Dept_No), synonym tables (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):

<table>
<thead>
<tr>
<th>Synthesized relationships</th>
<th>value-determined by DEPT_EMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dept# EPT_PROJECT</td>
<td>Dept#EMP_PROJECT</td>
</tr>
<tr>
<td>Dept# /* redundant */</td>
<td>PROJECT_TASK</td>
</tr>
<tr>
<td>Proj#EMP_TASK</td>
<td>Name /* meaningless */</td>
</tr>
</tbody>
</table>

(b) Infer maximum participations: If a value-determined relationship includes the key attributes of one entity, then maximum participation in that direction is 1.

Example: Maximum participation is 1 from EMP in DEPT_EMP; from PROJECT in DEPT_PROJECT; from TASK in PROJECT_TASK. Note that this is a rearrangement, introducing no new uncertainty.

c) Guess inclusion pattern from foreign key references: Suppose R is value-determined by matching E1[A1] and E2[A2], that 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] ... E2[A2].

Example: The following inclusions are guessed:

<table>
<thead>
<tr>
<th>DEPT[Dept#] ... EMP[Dept#]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPT[Dept#] ... PROJECT[Dept#] PROJECT[Proj#] ... TASK[Proj#]</td>
</tr>
</tbody>
</table>

Example: We do not guess inclusions for 1:1 relationships. To see why, consider a schema with three relations keyed on social security number:

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

Relationship creation for matching attributes (step a) will guess that each pair of entities has a relationship determined by SS#; constraint inference (step b) will infer that the relationships must have maximum participation of 1 in each direction. However, the schema provides no basis for guessing inclusions.

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.

(e) Delete relationships derived from nonsensical matches: While many relationships created by our heuristics are meaningful, some are based on nonsensical matches (e.g. TASK.Name and EMP.Name). DDEW has two mechanisms to bring such situations to the user's attention. First, relationships based on non-keys 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. We reasoned that mere exhortation to check tool results would be ineffective. Manual relationship deletion is then possible.

(f) Detect inadvertent omissions (where possible): DDEW includes a navigational, model-independent language in which designers were 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.

For schemas that already have relationships, steps a-f are 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

---

1 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.
Unnecessary relationships make a schema more cluttered and less understandable. In addition, because they may appear to impose constraints, they can inhibit rearrangements (e.g., removal of a foreign key attribute that determines the redundant relationship). In step (d), we delete relationships R such that:

1) **Deletion of R will produce a content-equivalent schema.** A value-determined relationship with no other constraints can be deleted without changing information content; the incident entities determine the relationship's population. (The constraints inferred at step (b) can always be removed by the inverse rearrangement.)

2) **R is provably the composition of other relationships**, so it is not establishing an important semantic connection (see the Composition Theorem below). For example, EMP_PROJECT can be deleted.

3) **R is not “semantically significant.”** If the relationship was manually inserted or the user has manually given it a name, this is a hint that the relationship may be significant, so deletion will require explicit approval.

Conditions (2) and (3) above are an attempt to judge the desirability of deletion rather than its correctness. In general models, composition of relationships is hard to determine, but value-based relationships make the task much easier. Note that the following theorem does not require that relationships be based on foreign keys.

**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.**
E1[A,A'] matches E2[B,B']

E2[B] matches E3[C]

R12
Proof: To show that $R_{13}$ equals the composition $R_{12}\circ R_{23}$, we show set containment in each direction.

(1) To prove $R_{12}\circ R_{23}$ is contained in $R_{13}$, suppose $(e_1,e_3) \in R_{12}\circ R_{23}$.

By definition of composition, $R_{12}\circ R_{23} = \{(e_1,e_3) | \exists e_2$ such that $(e_1,e_2) \in R_{12}$ and $(e_2,e_3) \in R_{23}\}$. Because $R_{12}$ is value-determined, $e_1[A,A'] = e_2[B,B']$. Hence, $e_1[A] = e_2[B]$.

Since $R_{23}$ is value-determined, $e_2[B] = e_3[C]$. Hence $e_1[A] = e_3[C]$, so $(e_1,e_3) \in R_{13}$.

(2) To prove $R_{12}\circ R_{23}$ contains $R_{13}$, suppose $(e_1,e_3) \in R_{13}$.

Then $e_1[A] = e_3[C]$, and $e_1[A]$ is nonnull.

Since $E_2[B,B'] \ldots E_1[A,A']$, $\exists e_2$ such that $e_2[B,B'] = e_1[A,A']$. Hence $e_2[B] = e_1[A]$, so $e_2[B] = e_3[C]$.

The equalities and the fact that $R_{12}$ and $R_{23}$ are value-determined imply that $(e_1,e_2) \in R_{12}$ and $(e_2,e_3) \in R_{23}$.

Hence $(e_1,e_3)$ is in $R_{12}\circ R_{23}$.

Example: "Horizontal" Shortcut

In Figure 5, the relationship EMP_PROJECT is redundant. EMP_PROJECT must be the composition of DEPT_PROJECT and DEPT_EMP (by the Composition Theorem, with $A=B=C=(\text{Dept#})$, $A'=B'=\Gamma$). Such horizontal relationships can substantially complicate a schema. If $n$ entity types reference DEPT using Dept# as a foreign key, then $n(n-1)/2$ redundant, “horizontal” relationships are generated.
Example: "Vertical" Shortcut
In Figure 6, the relationship between STATE and STREET is redundant. Recall that the Composition Theorem requires value-determined relationships and just one inclusion constraint – streets within a state must be in a city in that state. However, neither keys, nor participation constraints, nor other inclusions are required. That is, the database can contain (rural) streets with null city and/or state 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
Vertical shortcut rules have been used in design algorithms proposed previously [KLUG79, TEOR86]. But without value-determination, there is no formal information in the schema to enable a tool to determine whether a relationship is redundant. Full relational theory permits identification of redundant inclusions, but our conditions are more general. Consider the example in Figure 7 below, in which Teorey recommends that the relationship ATTENDS be dropped [TEOR86].
Segev observes that the above diagram does not itself determine whether **ATTENDS** is redundant [SEGE87]. He therefore asks the user to provide additional constraints, such as:

```
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. In ER+, the designer need only check on the determining attributes of each relationship individually. Information implicit in having value-determined relationships over the same attributes then disambiguates the situation.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Value-determined by</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUDENT(Stud#, Club#, School#, ...)</td>
<td>Club#</td>
</tr>
<tr>
<td>CLUB(Club#, School#, ...)</td>
<td>School#</td>
</tr>
<tr>
<td>SCHOOL(School#, ...)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Value-determined by</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATED_IN(CLUB,SCHOOL)</td>
<td>School#</td>
</tr>
<tr>
<td>ATTENDS(STUDENT,SCHOOL)</td>
<td>School#IS_MEMBER(STUDENT,CLUB)</td>
</tr>
<tr>
<td></td>
<td>Club#, School#</td>
</tr>
</tbody>
</table>

Both the value-determined constraint on **IS_MEMBER** and the inclusion constraint involve multiple attributes, so the designer can be notified to check them. 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 answer is affirmative.

Segev raises the issue of whether deleting redundant relationships hurts the robustness of the schema [SEGE87]. In the case of horizontal shortcuts (detectable by examining the pattern of keys) we suspect that it does not. In ER+, the conclusion about redundancy is fairly robust, since the Composition Theorem’s preconditions are so weak. Of course, an explicit change to the determining attributes can make a relationship non-redundant.

### 4.3 View Integration

The research literature on view integration (see [BATI86] for an overview) consists principally of careful case-by-case heuristics for resolving particular types of mismatches between schemas (e.g., conditionally-mergeable relationships [NAVA86]). A contrasting approach is given in [CASA83, BISK86], 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, acquaint the designer with the conflicts and potential conflict resolutions, and automatically construct 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 diverse views into a single ER schema. When applied recursively, this process will result in a single, global 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 below. 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-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 [NAVA86]). 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 cardinality 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 (non-matching) objects from the two 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 design 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 (1984), the only ER schema rearrangements we found in the literature were elimination of redundant constraints, and schema normalization based on functional dependencies [DATR84]. 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., [CASA84, MARK89]) 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 [KALI90], or offering the designer a choice of alternative logical transformations on a conceptual generalization hierarchy [OERT89]. There have also been improvements to underlying algorithms such as normalization [DIED88], to constraint specification [MARK90], and to graphical formulation of design semantics and process [SOCK88]. For an overview of current automated design tools, both research-oriented and commercial, see [REIN91].

In all 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.

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 in a meaningful way. The formulations in the mathematical literature naturally emphasize generality, elegance and mathematical convenience. But system-builders need – comprehensible on-screen forms 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, [LING86] proved that existing multi-valued 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 below.

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. In addition, this technique makes the system much 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 meaningful, but is just not accessible to 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.
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 non-rigorous 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 inter-model 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.
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.
7. References


**Acknowledgements**

V. Markowitz’s challenging comments encouraged us to clarify the justification of ER+.

Our heartfelt thanks go 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.
8 Appendix: Proofs for Section 3

This appendix contains proofs that the formal transformations in Section 3 are indeed rearrangements. We will first describe some general tactics that are used repeatedly.

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.

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.

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 as the easy case. That instance mapping is known to be injective. So one need only verify total and surjective. To verify that an injective mapping I is surjective, it is sufficient to define a mapping I⁻¹ that is the inverse of I. In other words, for any schema s2 satisfying S2, I⁻¹(s2) satisfies S1 and I(I⁻¹(s2)) = s2.

8.1 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[A1].
  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.

  Injectable: 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 e s1) but not in population(E2 e s1'). Then I(E2 e s1) includes an instance extending e2 with e1[A1], while I(E2 e s1') does not -- a contradiction.

8.2 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].

  The instantiation mapping is the same as for key-copying.

  Injectable: 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.

8.3 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. —

8.4 Multiset 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.

    $E1\rightarrow(m_1,M_1)\rightarrow<R12>\rightarrow(m_2,M_2)\rightarrow E2$ is rearranged to: $E1\rightarrow(m_1,M_1)\rightarrow<E13>\rightarrow(1,1)\rightarrow R13\rightarrow(1,1)\rightarrow E3\rightarrow(1,1)\rightarrow R23\rightarrow(m_2,M_2)\rightarrow E2$

  **Proof:**
  Define an instance map $I$ that leaves objects other than R12 unchanged. It populates E2, R13, and R23 by:
  - $\text{population}(E3) = \{ \text{one instance for each } R12 \text{ instance } (e1,e2) \}$
  - $\text{population}(R13) = \{ (e1,e3) | e3 corresponds to an R12 instance (e1,x) for some x \}$
  - $\text{population}(R23) = \{ (e2,e3) | e3 corresponds to an R12 instance (e2,x) for some x \}$

  **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 $\text{population}(R12)$ as the multiset containing an instance $(e1,e2)$ of R12 for each $(e1,e3)$ in R13 and $(e2,e3)$ 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 (labelled 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, $\text{population}(R12 \text{ in } s1)$) 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. —

8.5 Modification to deal with a set relationship

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

  We compose several other rearrangements to handle the set-relationship constraint. The steps are:
a. Identify primary keys K1 for E1, K2 for E2.
b. Execute the multiset rearrangement (above).
c. Copy keys K1 from E1 and K2 from E2 into E3.
d. 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~ 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~ satisfies the constraints of S1~. So it remains only to show that I~ maps S2 into S1, i.e., that the set-relationship constraint holds in I~(s2). So it remains only to show that I~ maps S2 into S1, i.e., that the set-relationship constraint holds in I~(s2).

I~ populates R12 by creating one instance of each triple (e1,e3,e2). For the set relationship constraint on R12 to be violated, S2 must contain e3 and e3' satisfying: [(e1,e3) e R13 and (e2,e3) e R23] and [(e1,e3') e R13 and (e2,e3') e 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.

### 8.6 Converting a value-determined set relationship

**Converting a value-determined set relationship R12 (between entities E1 and E2) 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.

\[
E1---(m1,1)---<R12>---(m2,2)---E2 \text{ is rearranged to: } \ E1---(m1^-,1)---
\]

\[
<R13>---(m2^+,2)---E3---(m1^+,1)---R23---(m2^-,1)---E2
\]

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 3.5) 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~ 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]…E1[A1], then all values from E1[A1] generated tuples of E3, so E3[A3]…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 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 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 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 set-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]…E1[A1] on S2. Since E2[A2]…E3[A3] we have E2[A2]…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.

8.7 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}

(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. —